

conference

proceedings

# USENIX 1997 Annual Technical Conference

*Anaheim, California*

*January 6-10, 1997*



The UNIX® and Advanced  
Computing Systems Professional  
and Technical Association

For additional copies of these proceedings contact:

USENIX Association  
2560 Ninth Street, Suite 215  
Berkeley, CA 94710 USA  
Phone: 510 528 8649  
FAX: 510 548 5738  
Email: [office@usenix.org](mailto:office@usenix.org)  
URL: <http://www.usenix.org>

The price is \$32 for members and \$40 for nonmembers.

Outside the U.S.A. and Canada, please add  
\$18 per copy for postage (via air printed matter).

### Past USENIX Technical Conferences

|                            |                            |
|----------------------------|----------------------------|
| 1996 San Diego             | 1989 Winter San Diego      |
| 1995 New Orleans           | 1988 Summer San Francisco  |
| 1994 Summer Boston         | 1988 Winter Dallas         |
| 1994 Winter San Francisco  | 1987 Summer Phoenix        |
| 1993 Summer Cincinnati     | 1987 Winter Washington, DC |
| 1993 Winter San Diego      | 1986 Summer Atlanta        |
| 1992 Summer San Antonio    | 1986 Winter Denver         |
| 1992 Winter San Francisco  | 1985 Summer Portland       |
| 1991 Summer Nashville      | 1985 Winter Dallas         |
| 1991 Winter Dallas         | 1984 Summer Salt Lake City |
| 1990 Summer Anaheim        | 1984 Winter Washington, DC |
| 1990 Winter Washington, DC | 1983 Summer Toronto        |
| 1989 Summer Baltimore      | 1983 Winter San Diego      |

1997 © Copyright by The USENIX Association  
All Rights Reserved.

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. USENIX acknowledges all trademarks appearing herein.

ISBN 1-880446-84-7

Printed in the United States of America on 50% recycled paper, 10-15% post consumer waste.

**The USENIX Association**

**Proceedings of the  
USENIX 1997 Annual Technical Conference**

**January 6-10, 1997  
Anaheim, California, USA**

## ACKNOWLEDGMENTS

### Program Chair

John Kohl, *Pure Atria Corporation*

### Program Committee

Matt Blaze, *AT&T Labs-Research*  
Bill Bolosky, *Microsoft Research*  
Nathaniel Borenstein, *First Virtual Holdings*  
Charlie Briggs, *Digital Equipment Corp.*  
Clem Cole, *Digital Equipment Corp.*  
Fred Douglass, *AT&T Labs-Research*  
Rob Gingell, *Sun Microsystems*  
Mike Karels, *Berkeley Software Design, Inc.*  
John Schimmel, *Silicon Graphics*  
Carl Staelin, *Hewlett-Packard Labs*

### External Reviewers

|                   |                     |
|-------------------|---------------------|
| Howard Alt        | Susan LoVerso       |
| Remzi Arpaci      | Paula Long          |
| Gaurav Banga      | Stephen Manley      |
| Trevor Blackwell  | Heidi McClure       |
| Aaron Brown       | Kirk McKusick       |
| Larry Cable       | Larry McVoy         |
| Karen A. Casella  | Ethan Miller        |
| Mike Dahlin       | Greg Minshall       |
| Jeff Deutch       | Robert Morris       |
| Andrea Dusseau    | Adam Moskowitz      |
| John Dustin       | Bill Nesheim        |
| Yasuhiro Endo     | George Neville-Neil |
| Billy Fuller      | Kelly O'Hair        |
| John L. Furlani   | John Ousterhout     |
| Kanth Ghatraju    | Kent Peacock        |
| Timothy Gibson    | Jayarami A. Reddy   |
| John Heidemann    | Jim Rees            |
| Will Hill         | John Roach          |
| Larry Huston      | Glenn Scott         |
| John Ioannidis    | Steve Senator       |
| Chris Jackson     | Chris Small         |
| Deepak Kakadia    | Keith Smith         |
| Dinesh Katiyar    | Ramesh Subrahmaniam |
| Dave Korn         | Eric Sultan         |
| Christopher Krebs | Win Treese          |
| Orran Krieger     | Amin Vahdat         |
| Tom LaStrange     | Linda Wang          |
| Diane Lebel       | Jim Woodward        |
| John LoVerso      | Cliff Young         |

### Invited Talks and Guru-Is-In Coordinators

Mary Baker, *Stanford University*  
Barry Kercheval, *Xerox Parc*

### Works in Progress Coordinator

John Schimmel, *Silicon Graphics*

### USENIX Board Liaison

Daniel E. Geer, Jr., *Open Market, Inc.*

### Terminal Room

Gretchen Phillips, *University at Buffalo*

### Administration: USENIX Association Staff

Ellie Young, *Executive Director*  
Judy DesHarnais, *Meeting Planner*  
Daniel V. Klein, *Tutorial Director*  
Zanna Knight, *Marketing*  
Cynthia Deno, *Vendor Exhibition*

### USENIX Proceedings Production

Pennfield Jensen, *Publications Manager*  
Data Reproductions

### USENIX Support Staff

Colleen Biddle  
Eileen Curtis  
Diane DeMartini  
Julie Keiser  
Toni Veglia

# CONTENTS

|                    |    |
|--------------------|----|
| Preface .....      | v  |
| Author Index ..... | vi |

## Wednesday, January 8

### Performance I

*Session Chair: Carl Staelin, Hewlett-Packard Laboratories*

|  |    |
|--|----|
| Embedded Inodes and Explicit Grouping: Exploiting Disk Bandwidth for Small Files .....     | 1  |
| <i>Gregory R. Ganger, M. Frans Kaashoek, Massachusetts Institute of Technology</i>         |    |
| Observing the Effects of Multi-Zone Disks .....  | 19 |
| <i>Rodney Van Meter, Information Sciences Institute, University of Southern California</i> |    |
| A Revisitation of Kernel Synchronization Schemes .....                                     | 31 |
| <i>Christopher Small, Stephen Manley, Harvard University</i>                               |    |

### Interface Tricks

*Session Chair: Rob Gingell, Sun Microsystems*

|  |    |
|--|----|
| Porting UNIX to Windows NT .....   | 43 |
| <i>David G. Korn, AT&amp;T Labs-Research</i>   |    |
| Protected Shared Libraries—A New Approach to Modularity and Sharing .....  | 59 |
| <i>Arindam Banerji, Hewlett-Packard Laboratories, John M. Tracey, T.J. Watson Research Center;<br/>David L. Cohn, University of Notre Dame</i> |    |
| Extending the Operating System at the User-Level: the Ufo Global File System .....   | 77 |
| <i>Albert D. Alexandrov, Maximilian Ibel, Klaus E. Schauser, Chris J. Scheiman, University of<br/>California, Santa Barbara</i>                |    |

### Client Tricks

*Session Chair: Fred Douglass, AT&T Labs-Research*

|  |     |
|--|-----|
| Network-aware Mobile Programs .....  | 91  |
| <i>Mudumbai Ranganathan, Anurag Acharya, Shamik Sharma, Joel Saltz, University of Maryland</i>                                     |     |
| Using Smart Clients to Build Scalable Services .....   | 105 |
| <i>Chad Yoshikawa, Brent Chun, Paul Eastham, Amin Vahdat, Thomas Anderson, David Culler<br/>University of California, Berkeley</i> |     |

## Thursday, January 9

### Clustering

*Session Chair: Clem Cole, Digital Equipment Corporation*

|   |     |
|---|-----|
| Building Distributed Process Management on an Object-Oriented Framework .....                                       | 119 |
| <i>Ken Shirriff, Sun Microsystems Laboratories</i>  |     |
| Adaptive and Reliable Parallel Computing on Networks of Workstations .....  | 133 |
| <i>Robert D. Blumofe, University of Texas, Austin; Philip A. Lisecki, Massachusetts<br/>Institute of Technology</i> |     |
| A Distributed Shared Memory Facility for FreeBSD .....  | 149 |
| <i>Pedro A. Souto; Eugene W. Stark, State University of New York, Stony Brook</i>                                   |     |

## Tools

*Session Chair: Matt Blaze, AT&T Labs-Research*

|  |     |
|--|-----|
| Cdt: A General and Efficient Container Data Type Library . . . . .         | 163 |
| <i>Kiem-Phong Vo, AT&amp;T Labs-Research</i>                               |     |
| A Simple and Extensible Graphical Debugger . . . . .                       | 173 |
| <i>David R. Hanson, Jeffrey L. Korn, Princeton University</i>              |     |
| Cget, Cput, and Stage—Safe File Transport Tools for the Internet . . . . . | 185 |
| <i>Bill Cheswick, Bell Laboratories</i>                                    |     |

## Friday, January 10

### User Tools

*Session Chair: Charlie Briggs, Digital Equipment Corporation*

|   |     |
|---|-----|
| WebGlimpse—Combining Browsing and Searching . . . . .                               | 195 |
| <i>Udi Manber, Michael Smith, Burra Gopal, University of Arizona</i>                |     |
| Mailing List Archive Tools . . . . .  | 207 |
| <i>Sam Leffler, Silicon Graphics; Melange Tortuba, Tortuba Consulting</i>           |     |
| Experience with GroupLens: Making Usenet Useful Again . . . . .                     | 219 |
| <i>Bradley N. Miller; John T. Riedl; Joseph A. Konstan, University of Minnesota</i> |     |

### Performance II

*Session Chair: Mike Karels, Berkeley Software Design*

|   |     |
|---|-----|
| Overcoming Workstation Scheduling Problems in a Real-Time Audio Tool . . . . .                      | 235 |
| <i>Isidor Kouvelas, Vicky Hardman, University College London</i>                                    |     |
| On Designing Lightweight Threads for Substrate Software . . . . .                                   | 243 |
| <i>Matthew Haines, University of Wyoming</i>  |     |
| High-Performance Local Area Communication With Fast Sockets . . . . .                               | 257 |
| <i>Steven H. Rodrigues, Thomas E. Anderson, David E. Culler, University of California, Berkeley</i> |     |

### Caching and Stashing

*Session Chair: Bill Bolosky, Microsoft Research*

|   |     |
|---|-----|
| An Analytical Approach to File Prefetching . . . . .  | 275 |
| <i>Hui Lei, Dan Duchamp, Columbia University</i>  |     |
| Optimistic Deltas for WWW Latency Reduction . . . . .   | 289 |
| <i>Gaurav Banga, Rice University; Fred Douglass, Michael Rabinovich, AT&amp;T Labs-Research</i> |     |
| A Toolkit Approach to Partially Connected Operation . . . . .                                   | 305 |
| <i>Dan Duchamp, Columbia University</i>   |     |

# Preface

Welcome to Anaheim and to the 1997 USENIX Conference!

On behalf of the USENIX Association, thank you for coming to the conference. We have a lot going on here this year: the refereed paper tracks, invited talks, works-in-progress and guru-is-in sessions, and two full days of timely tutorials. New at the 1997 conference is the co-location of and joint registration with the USELINUX conference. We think you'll find plenty of interesting sessions here to keep you busy the whole week!

We have chosen 23 papers for the refereed track this year. They cover topics such as filesystems, networking, networked systems, programming tools, user tools, and performance. Please extend my thanks to the 155 authors from 37 universities, 13 companies, and 11 countries for their 74 paper submissions for the technical track. Without their papers describing their recent efforts, we would not have a refereed track to present. Some of the papers not accepted for the refereed track will appear informally, in the works-in-progress session or maybe even in a birds-of-a-feather gathering.

There are many people involved in preparing a USENIX conference, many more than space here can mention. Those of special note include the wonderful USENIX staff I worked with: Ellie Young, Judy DesHarnais, Zanna Knight, and Pennfield Jensen—they were always glad to offer me advice, direction, and able assistance. Dan Geer, my USENIX board liaison, kept a watchful eye and was always ready to listen. Special thanks go to Mary Baker and Berry Kercheval for arranging the invited talks sessions; to Dan Klein who organized the fantastic tutorial selection; to Margo Seltzer and Vera Gropper at Harvard University for hosting the program committee meeting; to Keith Smith who served as the Program Committee's scribe; and to my employer, Pure Atria, for supporting my work as chair.

Finally, I would like to thank my program committee (10 hardy souls) and the external reviewers (listed separately) for their long hours reading papers and writing detailed reviews last summer. The program committee provided feedback to the authors of each submission, drawing from the 5 or more reviews of each paper. We've got a fine technical track this year due to the hard work and hard decisions of these volunteers.

John Kohl, Program Chair

# Author Index

|                           |          |                            |     |
|---------------------------|----------|----------------------------|-----|
| Anurag Acharya.....       | 91       | Sam Leffler .....          | 207 |
| Albert D. Alexandrov..... | 77       | Philip A. Lisiecki.....    | 133 |
| Thomas E. Anderson .....  | 105, 257 | Udi Manber .....           | 195 |
| Gaurav Banga.....         | 289      | Stephen Manley.....        | 31  |
| Arindam Banerji.....      | 59       | Bradley N. Miller .....    | 219 |
| Robert D. Blumofe.....    | 133      | Michael Rabinovich .....   | 289 |
| Bill Cheswick.....        | 185      | Mudumbai Ranganathan ..... | 91  |
| Brent Chun.....           | 105      | John T. Riedl .....        | 219 |
| David L. Cohn .....       | 59       | Steven H. Rodrigues .....  | 257 |
| David E. Culler.....      | 105, 257 | Joel Saltz .....           | 91  |
| Fred Douglass.....        | 289      | Klaus E. Schauer .....     | 77  |
| Dan Duchamp .....         | 275, 305 | Chris J. Scheiman .....    | 77  |
| Paul Eastham .....        | 105      | Shamik Sharma .....        | 91  |
| Gregory R. Ganger.....    | 1        | Ken Shirriff .....         | 119 |
| Burra Gopal .....         | 195      | Christopher Small .....    | 31  |
| Matthew Haines .....      | 243      | Michael Smith.....         | 195 |
| David R. Hanson .....     | 173      | Pedro A. Souto .....       | 149 |
| Vicky Hardman .....       | 235      | Eugene W. Stark .....      | 149 |
| Maximilian Ibel .....     | 77       | Melange Tortuba .....      | 207 |
| M. Frans Kaashoek .....   | 1        | John M. Tracey .....       | 59  |
| Joseph A. Konstan .....   | 219      | Amin Vahdat .....          | 105 |
| David G. Korn .....       | 43       | Rodney Van Meter.....      | 19  |
| Jeffrey L. Korn.....      | 173      | Kiem-Phong Vo.....         | 163 |
| Isidor Kouvelas .....     | 235      | Chad Yoshikawa .....       | 105 |
| Hui Lei.....              | 275      |                            |     |

# Embedded Inodes and Explicit Grouping: Exploiting Disk Bandwidth for Small Files

Gregory R. Ganger and M. Frans Kaashoek  
M.I.T. Laboratory for Computer Science  
Cambridge MA 02139, USA  
{ganger, kaashoek}@lcs.mit.edu  
<http://www.pdos.lcs.mit.edu/>

## Abstract

*Small file performance in most file systems is limited by slowly improving disk access times, even though current file systems improve on-disk locality by allocating related data objects in the same general region. The key insight for why current file systems perform poorly is that locality is insufficient — exploiting disk bandwidth for small data objects requires that they be placed adjacently. We describe C-FFS (Co-locating Fast File System), which introduces two techniques, **embedded inodes** and **explicit grouping**, for exploiting what disks do well (bulk data movement) to avoid what they do poorly (reposition to new locations). With embedded inodes, the inodes for most files are stored in the directory with the corresponding name, removing a physical level of indirection without sacrificing the logical level of indirection. With explicit grouping, the data blocks of multiple small files named by a given directory are allocated adjacently and moved to and from the disk as a unit in most cases. Measurements of our C-FFS implementation show that embedded inodes and explicit grouping have the potential to increase small file throughput (for both reads and writes) by a factor of 5–7 compared to the same file system without these techniques. The improvement comes directly from reducing the number of disk accesses required by an order of magnitude. Preliminary experience with software-development applications shows performance improvements ranging from 10–300 percent.*

## 1 Introduction

It is frequently reported that disk access times have not kept pace with performance improvements in other system components. However, while the time required to fetch the first byte of data is high (i.e., measured in mil-

liseconds), the subsequent data bandwidth is reasonable (> 10 MB/second). Unfortunately, although file systems have been very successful at exploiting this bandwidth for large files [Peacock88, McVoy91, Sweeney96], they have failed to do so for small file activity (and the corresponding metadata activity). Because most files are small (e.g., we observe that 79% of all files on our file servers are less than 8 KB in size) and most files accessed are small (e.g., [Baker91] reports that 80% of file accesses are to files of less than 10KB), file system performance is often limited by disk access times rather than disk bandwidth.

One approach often used in file systems like the fast file system (FFS) [McKusick84] is to place related data objects (e.g., an inode and the data blocks it points to) near each other on disk (e.g., in the same cylinder group) in order to reduce disk access times. This approach can successfully reduce the seek time to just a fraction of that for a random access pattern. Unfortunately, it has some fundamental limitations. First, it affects only the seek time component of the access time<sup>1</sup>, which generally comprises only about half of the access time even for random access patterns. Rotational latency, command processing, and data movement, which are not reduced by simply placing related data blocks in the same general area, comprise the other half. Second, seek times do not drop linearly with seek distance for small distances. Seeking a single cylinder (or just switching between tracks) generally costs a full millisecond, and this cost rises quickly for slightly longer seek distances [Worthington95]. Third, it is successful only when no other activity moves the disk arm between related requests. As a result, this approach is generally limited to providing less than a factor of two improvement in performance (and often much less).

Another approach, the log-structured file system

<sup>1</sup>We use the terms *access time* and *service time* interchangeably to refer to the time from when the device driver initiates a read or write request to when the request completion interrupt occurs.

(LFS), exploits disk bandwidth for all file system data, including large files, small files, and metadata. The idea is to delay, remap and cluster all modified blocks, only writing large chunks to the disk [Rosenblum92]. Assuming that free extents of disk blocks are always available, LFS works extremely well for write activity. However, the design is based on the assumption that file caches will absorb all read activity and does not help in improving read performance. Unfortunately, anecdotal evidence, measurements of real systems (e.g., [Baker91]), and simulation studies (e.g., [Dahlin94]) all indicate that main memory caches have not eliminated read traffic.

This paper describes the co-locating fast file system (C-FFS), which introduces two techniques for exploiting disk bandwidth for small files and metadata: *embedded inodes* and *explicit grouping*. Embedding inodes in the directory that names them (unless multiple directories do so), rather than storing them in separate inode blocks, removes a physical on-disk level of indirection without sacrificing the logical level of indirection. This technique offers many advantages: it halves the number of blocks that must be accessed to open a file; it allows the inodes for all names in a directory to be accessed without requesting additional blocks; it eliminates one of the ordering constraints required for integrity during file creation and deletion; it eliminates the need for static (over-)allocation of inodes, increasing the usable disk capacity [Forin94]; and it simplifies the implementation and increases the efficiency of explicit grouping (there is a synergy between these two techniques).

Explicit grouping places the data blocks of multiple files at adjacent disk locations and accesses them as a single unit most of the time. To decide which small files to co-locate, C-FFS exploits the inter-file relationships indicated by the name space. Specifically, C-FFS groups files whose inodes are embedded in the same directory. The characteristics of disk drives have reached the point that accessing several blocks rather than just one involves a fairly small additional cost. For example, even assuming minimal seek distances, accessing 16 KB requires only 10% longer than accessing 8 KB, and accessing 64 KB requires less than twice as long as accessing a single 512-byte sector. Further, the relative cost of accessing more data has been dropping over the past several years and should continue to do so. As a result, explicit grouping has the potential to improve small file performance by an order of magnitude over conventional file system implementations. Because the incremental cost is so low, grouping will improve performance even when only a fraction of the blocks in a group are needed.

Figure 1 illustrates the state-of-the-art and the improvements made by our techniques. Figure 1A shows

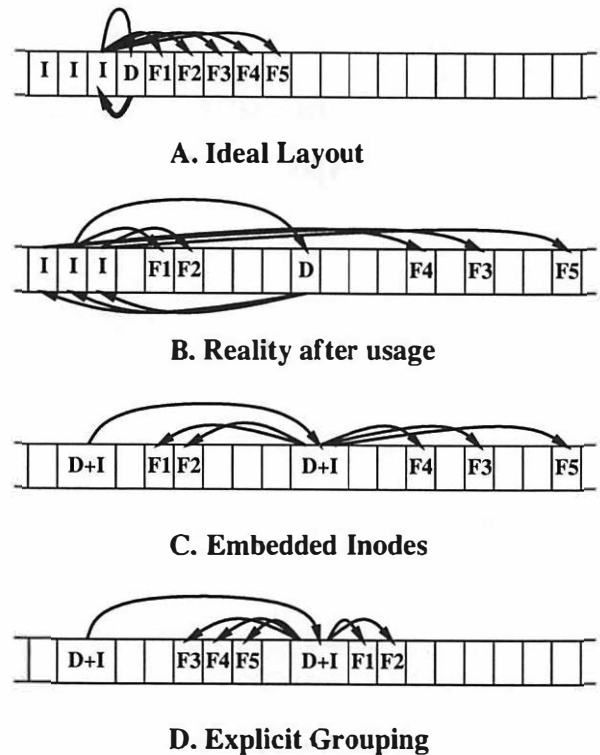


Figure 1: Organization and layout of file data on disk. This figure shows the on-disk locations of directory blocks (marked 'D'), inode blocks ('I') and the data for five single-block files ('F1' - 'F5') in four different scenarios: (A) the ideal conventional layout, (B) a more realistic conventional layout, (C) with the addition of embedded inodes, and (D) with both embedded inodes and explicit grouping (with a maximum group size of four blocks).

the ideal layout of data and metadata for five single-block files, which might be obtained if one uses a fresh FFS partition. In this case, the inodes for all of the files are located in the same inode block and the directory block and the five file blocks are stored adjacently. With this layout, the prefetching performed by most disks will exploit the disk's bandwidth for reads and scatter/gather I/O from the file cache can do so for writes. Unfortunately, a more realistic layout of these files for an FFS file system that has been in use for a while is more like that shown in Figure 1B. Reading or writing the same set of files will now require several disk accesses, most of which will require repositioning (albeit with limited seek distances, since the picture shows only part of a single cylinder group). With embedded inodes, one gets the layout shown in Figure 1C, wherein the indirection between on-disk directory entries and on-disk inodes is eliminated. Finally, with both embedded inodes and ex-

PLICIT grouping, one gets the layout shown in Figure 1D. In this case, one can read or write all five files with two disk accesses. Further, for files F1 and F2, one can read or write the name, inode and data block all in a single disk request. In our actual implementation, the maximum group size is larger than that shown and the allocation code would try to place the second group immediately after the first.

We have constructed a C-FFS implementation that includes both of these techniques. Measurements of C-FFS as compared to the same file system without these techniques show that, for small file activity, embedded inodes and explicit grouping can reduce the number of disk accesses by more than an order of magnitude. On the system under test (a modern PC), this translates into a performance increase of a factor of 5–7 in small file throughput for both reads and writes. Although our evaluation is preliminary, experiments with actual applications show performance improvements ranging from 10–300 percent.

The remainder of this paper is organized as follows: Section 2 provides more detailed motivation for co-location, by exploring the characteristics of modern disk drives and file storage usage. Section 3 describes our implementation of embedded inodes and explicit grouping. Section 4 shows that co-location improves performance significantly by comparing two file system implementations that differ only in this respect. Section 5 discusses related work. Section 6 discusses some open questions. Section 7 summarizes this paper.

## 2 Motivation

The motivating insights for this work fall into two broad categories: (1) The performance characteristics of modern disk drives, the de facto standard for on-line storage, force us to aggressively pursue adjacency of small objects rather than just locality. (2) The usage and organizational characteristics of popular file systems both suggest the logical relationships that can be exploited and expose the failure of current approaches in placing related data adjacently.

### 2.1 Modern Disk Drive Characteristics

It has repeatedly been pointed out that disk drive access times continue to fall behind relative to other system components. However, disk drive manufacturers have not been idle. They have matched the improvement rates of other system components in areas other than the access time, such as reliability, cost per byte, and recording density. Also, although it has not improved quite as rapidly, bulk data bandwidth has improved significantly.

This section uses characteristics of modern disk drives to show that reading or writing several 4KB or 8KB disk blocks costs a relatively small amount more than reading or writing only one (e.g., 10% for 8KB extra and 100% for 56KB extra), and that this incremental cost is dropping. Because of the high cost of accessing a single block, it makes sense to access several even if only some of them are likely to be necessary. Readers who do not need to be convinced are welcome to skip to Section 2.2.

The service time for a disk request can be broken into two parts, one that is dependent on the amount of data being transferred and one that is not. The former usually consists of just the media transfer time (unless the media transfer occurs separately, as with prefetch or write-behind). Most disks overlap the bus transfer time with the positioning and media transfer times, such that what remains, the ramp-up and ramp-down periods, is independent of the size. Most of the service time components, most notably including command processing overheads, seek times, and rotational latencies, are independent of the request size. With modern disk drives and small requests, the size-independent parts of the service time dominate the size-dependent part (e.g., > 90% of the total for random 8KB requests).

Another way to view the same two aspects of the service time are as per-request and per-byte. The dominating performance characteristic of modern disk drives is that the per-request cost is much larger than the per-byte cost. Therefore, transferring larger quantities of useful data in fewer requests will result in a significant performance increase.

One approach currently used by many file system implementors is to try to place related data items close to each other on the disk, thereby reducing the per-request cost. This approach does improve performance, but not nearly as much as one might hope. It is generally limited to reducing seek distances, and thereby seek times, which represent only about half of the per-request time. Rotational latency, command processing and data movement comprise the other half. In addition, even track switches and single-cylinder seeks require a significant amount of time (e.g., a millisecond or more) because they still involve mechanical movement and settling delays. Further, this approach is successful only when no other activity uses the disk (and thereby moves the disk arm) between related requests. As a result, this approach is generally limited to providing less than a factor of two in performance (and in practice much less).

To help illustrate the above claims, Table 1 lists characteristics for three state-of-the-art (for 1996) disk drives [HP96, Quantum96, Seagate96]. Figure 2 shows, for the same three drives, average access times as a function of the request size. Several points can be inferred from these graphs. First, the incremental cost of reading or

| Disk Drive Specification | Hewlett-Packard C3653a | Seagate Barracuda | Quantum Atlas II |
|--------------------------|------------------------|-------------------|------------------|
| Capacity                 | 8.7 GB                 | 9.1 GB            | 9.1 GB           |
| Cylinders                | 5371                   | 5333              | 5964             |
| Surfaces                 | 20                     | 20                | 20               |
| Sectors per Track        | 124-173                | 153-239           | 108-180          |
| Rotation Speed           | 7200 RPM               | 7200 RPM          | 7200 RPM         |
| Head Switch              | < 1 ms                 | N/A               | N/A              |
| One-cyl. Seek            | < 1 ms                 | 0.6 ms (0.5 ms)   | 1.0 ms           |
| Average Seek             | 8.7 ms (0.8 ms)        | 8.0 ms (1.5 ms)   | 7.9 ms           |
| Maximum Seek             | 16.5 ms (1.0 ms)       | 19.0 ms (1.0 ms)  | 18.0 ms          |

Table 1: Characteristics of three modern disk drives, taken from [HP96, Seagate96, Quantum96]. N/A indicates that the information was not available. For the seek times, the additional time needed for write settling is shown in parentheses, if it was available.

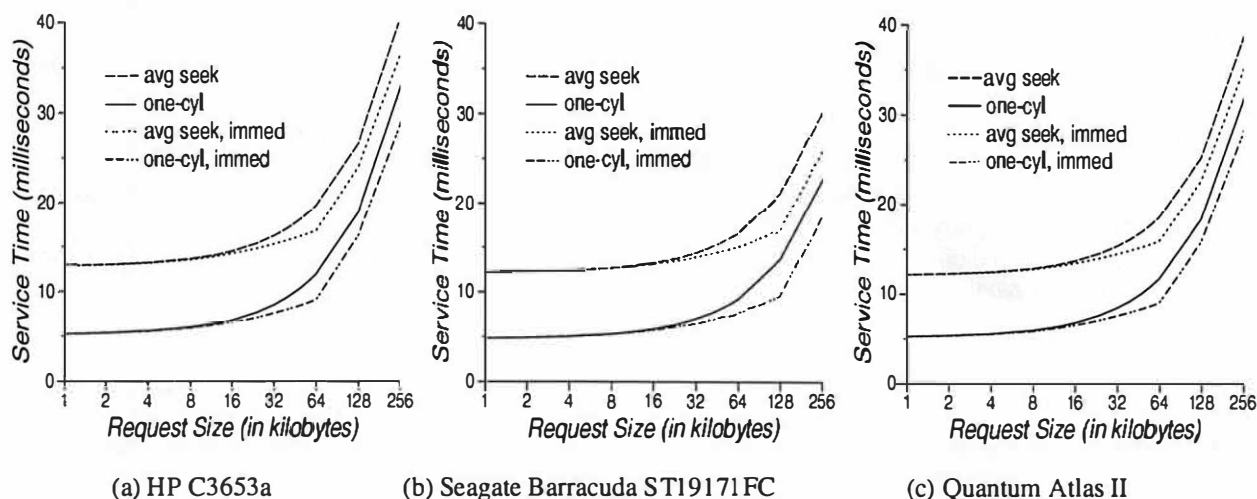


Figure 2: Average service time as a function of request size. The X-axis uses log scale. Two of the lines in each plot represent the access time assuming an average seek time and a single-cylinder seek time, respectively. The other two lines in each plot (labelled “immed”) represent the same values, but assuming that the disk utilizes a technique known as immediate or zero-latency access. The basic idea is to read or write the data sectors in the order that they pass under the read/write head rather than in strictly ascending order. This can eliminate part or all of the rotational latency aspect of the service time. These values were calculated from the data in Table 1 using the read seek times (which are shorter), assuming zero command initiation/processing overheads, and using the sector/track value for the outermost zone.

writing several blocks rather than just one is small. For example, a 16 KB access takes less than 10% longer than an 8 KB access, even assuming a minimal seek time. A 64 KB access takes less than twice as long as an 8 KB access. Second, immediate or zero-latency access extends the range of request sizes that can be accessed for small incremental cost, which will increase the effectiveness

of techniques that co-locate multiple small data objects. Third, the seek time for random requests represents a little more than half the total service time. Therefore, eliminating it completely for a series of requests can halve each service time after the first. In comparison, co-location will eliminate the entire service time for a set of requests after paying only a slightly higher cost

for the first. So, while reducing seek times can improve performance somewhat, aggressive co-location has the potential to provide much higher returns.

Not only are per-byte costs small relative to per-request costs, but they have been decreasing and are likely to continue to do so. For example, the HP C2247 disk drive [HP91, HP92] of a few years ago had only half as many sectors on each track as the HP C3653 listed in Table 1, but an average access time that was only 33% higher. As a result, a request of 64KB, which takes only 2 times as long as a request of 8KB on state-of-the-art disks, took nearly 3 times as long as a request of 8KB on older drives. Projecting this trend into the future suggests that co-location will become increasingly important.

## 2.2 File System Characteristics

File systems use metadata to organize raw storage capacity into human-readable name spaces. Most employ a hierarchical name space, using directory files to translate components of a full file name to an identifier for either the desired file or the directory with which to translate the next name component. At each step, the file system uses the identifier (e.g., an inode number in UNIX file systems) to determine the location of the file's metadata (e.g., an inode). This metadata generally includes a variety of information about the file, such as the last modification time, the length, and pointers to where the actual data blocks are stored. This organization involves several levels of indirection between a file name and the corresponding data, each of which can result in a separate disk access. Additionally, the levels of indirection generally cause each file's data to be viewed as an independent object (i.e., logical relationships between it and other files are only loosely considered).

To get a better understanding of real file storage organizations, we constructed a small utility to scan some of our group's file servers. The two SunOS 4.1 servers scanned supply 13.8 GB of storage from 9 file systems on 5 disks. At the time of the examination, 9.8 GB (71% of the total available) was allocated to 466,271 files.

In examining the statistics collected, three observations led us to explore embedded inodes and explicit grouping. First, as shown in Figure 3 as well as by previous studies, most files are small (i.e., 79% of those on our server are smaller than a single 8KB block). In addition to this static view of file size distributions, studies of dynamic file system activity have reported similar behavior. For example, [Baker91] states that, although most of the bytes accessed are in large files, "the vast majority of file accesses are to small files" (e.g., about 80% of the files accessed during their study were less than 10KB). These data, combined with the fact that

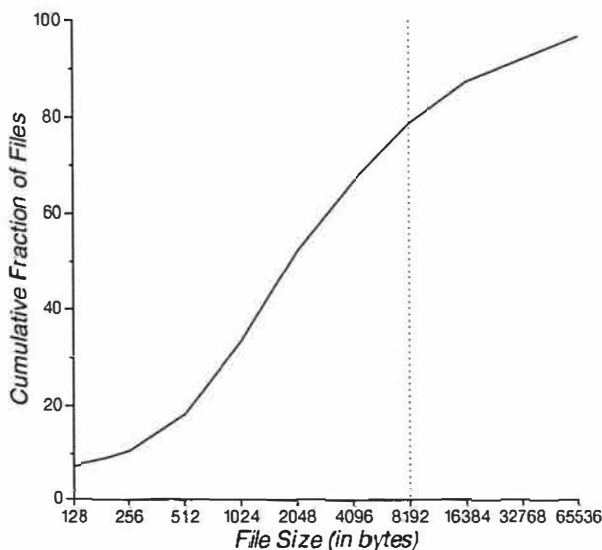


Figure 3: Distribution of file sizes measured on our file servers.

file system implementors are very good at dealing with large files, suggest that it is important to address the performance of small file access.

Second, despite the fact that each inode block contains 64 inodes, multiple inode blocks are often used to hold the metadata for files named in a particular directory (see Figure 4). On average, we found that the ratio of names in a directory to referenced inode blocks is roughly six to one. That is, every sixth name translates to a different inode block. This suggests that the current mechanism for choosing where to place an inode (i.e., inode allocation) does a poor job of placing related inodes adjacently. With embedded inodes, we store inodes in directories rather than pointers to inodes, except in the rare case (less than 0.1 percent, on our server) of having multiple links to a file from separate directories. In addition to eliminating a physical level of indirection, embedding inodes reduces the number of blocks that contain metadata for the files named by a particular directory.

Third, despite the fact that the allocation algorithm for single-block files in a directory might be expected to achieve an ideal layout (as in Figure 1A) on a brand new file system, these blocks tend to be local (e.g., in the same cylinder group) but not adjacent after the file system has been in use. For example, looking at entries in their physical order in the directory, we find that only 30% of directories are adjacent to the first file that they name. Further, fewer than 45% of files are placed adja-

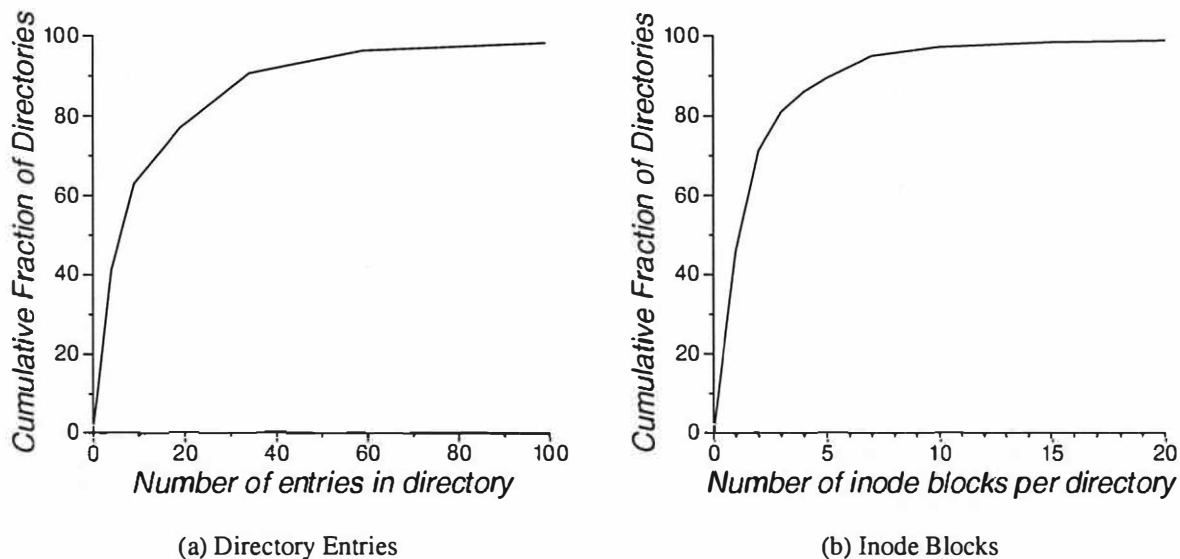


Figure 4: Distributions of the number of entries per directory and the corresponding number of inode blocks referred to on our file servers.

cent to the file whose name precedes their name in the directory.<sup>2</sup> The consequence of this lack of adjacency is that accessing multiple files in a directory involves disk head movement for every other file (when the file cache does not capture the accesses). The lack of adjacency occurs because, for the first block of a file, the file system always begins looking for free space at the beginning of the cylinder group. With explicit grouping, new file blocks for small files will be placed adjacently to other files in the same directory whenever possible. This should represent a substantial reduction in the number of disk requests required for a set of files named by a given directory.

We believe that the relationships inherent in the name space can be exploited to successfully realize the performance potential of modern disks' bandwidths. For example, many applications examine the attributes (i.e., reference the inodes) of files named by a given directory soon after scanning the directory itself (e.g., revision control systems that use modification times to identify files in the source tree that might have changed and di-

rectory listing programs that format the output based on permissions). Such temporal locality can be exploited directly with embedded inodes. Similarly, it is common for applications to access multiple files in a directory (e.g., when scanning a set of files for a particular string or when compiling and linking a multi-file program), as opposed to accessing files randomly strewn throughout the name space. Therefore, explicitly grouping small files in a directory represents a significant performance advantage.

### 3 Design and Implementation

To better exploit the data bandwidth provided by modern disk drives, C-FFS uses embedded inodes and explicit grouping to co-locate related small data objects. This section describes the various issues that arise when applying these techniques and how our C-FFS implementation addresses them.

#### 3.1 Embedded Inodes

Conceptually, embedding inodes in directories is straightforward. One simply eliminates the previous inode storage mechanism and replaces the inode pointer generally found in each directory entry with the inode itself. Unfortunately, a few complications do arise: (1) Finding the location of an arbitrary inode given an

<sup>2</sup>Different applications access files named by a directory in different orders. For example, many shell utilities re-order file listings alphabetically, while multi-file compile and link programs often apply a user-specified ordering. As a result, we also looked at file adjacency under alternative file sequences. For example, in alphabetical order, only 16% of directories are adjacent to their first file and only 30% of files are adjacent to their predecessor. Even with the best-case ordering (by ascending disk block address), only 64% of files are next to their predecessor.

inode number (and to avoid changing other file system components, this file ID must remain unique) must still be both possible and efficient. (2) As part of enabling the discovery of an arbitrary inode's location, the current C-FFS implementation replaces inode numbers with inode *locators* that include an ID for the containing directory. One consequence of this approach is that when a file is moved from one directory to another, its inode locator will change. This can cause difficulties for system components that are integrated with the file system and rely on inode number constancy. (3) Although files with links from multiple directories are rare, they are useful and should still be supported. Also, files with multiple links from within a directory should be supported with low cost, since some applications (such as editors) use such links for backup management. (4) File system recovery after both system crashes and partial media corruption should be no less feasible than with a conventional file system organization.

One of our implementation goals (on which we were almost successful) was to change nothing beyond the directory entry manipulation and on-disk inode access portions of the file system. This section describes how C-FFS implements embedded inodes, including how C-FFS addresses each of the above issues.

### Finding specific inodes

By eliminating the statically allocated, directly indexed set of inodes, C-FFS breaks the direct translation between inode number and on-disk inode location. To regain this, C-FFS replaces conventional inode numbers with inode *locators* and adds an additional on-disk data structure, called the *directory map*. In our current implementation, an inode locator has three components: a directory number, a sector number within the directory and an identifier within the sector. C-FFS currently sizes these at 16 bits, 13 bits and 3 bits, respectively, providing more than an order of magnitude more room for growth in both directory count and directory size than appears to be necessary from the characteristics of our file server. However, shifting to a 64-bit inode locator would overcome any realistic limitations.

To find the inode for an arbitrary inode locator, C-FFS uses the following procedure:

1. Look in the inode cache.
2. If the inode is not in the cache, use the directory number portion of the inode locator to index into the directory map. This provides the inode locator for the directory. A special directory number, 0, is used to refer to the "directory" containing multi-linked inodes that are not embedded (see below).

3. Get the directory's inode. This may require recursively executing steps 1, 2, and 3 several times. However, the recursion is always bounded by the root of the directory hierarchy and should rarely occur in practice, since getting to the file's inode locator in the first place required looking in the directory. Further, this recursion could be eliminated by exploiting the auxiliary physical location information included in the directory map (see below).
4. Read the directory block that contains the sector referred to by the sector number component of the inode number. (Note that the sector number is not really necessary — it simply allows us to restrict the linear scan.)
5. Traverse the set of directory entries in the sector to find the matching inode information, if it exists.

### Moving files

Because C-FFS stores inodes inside the directories that name them, moving a file from one directory to another involves also moving the inode. Fortunately, this is easy. However, because inode locators encode the inode's location, including the containing directory's number, the inode locator must be changed when an inode moves.<sup>3</sup> In addition to file movement, C-FFS moves inodes when they are named by multiple directories (see below) and when the last name for a file is removed while it is still open (to support POSIX semantics). Unfortunately, changing the externally visible file ID can cause problems for system components, such as an NFS server, that relies on constancy of the file ID for a given file. Some solutions to this problem that we have considered include not actually moving the inode (and using the external link entry type discussed below), keeping an additional structure to correlate the old and new inode locators and forcing other system components to deal with the change. We currently use this last solution, but are growing concerned that too many applications expect file IDs to remain constant.

We currently believe that the correct solution is to bring back constant inode numbers and to introduce another table to translate inode numbers to inode locators. Properly implemented, we believe that such a table could be maintained with low cost. In particular, it would only need to be read when a desired inode is not in the inode cache. Also, by keeping both the inode number and the

<sup>3</sup>Note that moving a directory will change the inode locator for the directory but will not change the inode locators for any of the files it names. The inode locators for these latter files encode the directory's identity by its index into the directory map rather than by its inode locator.

inode locator in each inode, the table could be considered soft state that is reconstructed if the system crashes.

### Supporting hard links

Although it is rare to have multiple hard links to a single file, it can be useful and we want to continue to allow it. Therefore, C-FFS replaces the inode pointer field of conventional directories with a type that can take on one of five values: (1) *invalid*, which indicates that the directory entry is just a space holder; (2) *embedded*, which indicates that the inode itself is in the entry; (3) *directory pointer*, which indicates that the entry contains a directory number that can be used to index into the directory map to find the corresponding inode locator (this entry type is used only for the special '.' and '..' entries); (4) *internal link*, which indicates that the entry contains a pointer to the location of the actual inode elsewhere within the directory; and (5) *external link*, which says that the entry contains a pointer to the location of the inode outside of the directory. Additional space in the directory entry is used to hold the inode or pointer for the latter four cases.

As suggested by the existence of an external link type, C-FFS stores some inodes outside of directories. In particular, C-FFS does this for inodes that are named by multiple files and for inodes that are not named but have been opened by some process. For the latter case, externalizing an inode is simple because we don't need to guarantee its existence across a system failure. Externalizing an inode to add a second external link, on the other hand, is expensive in our current C-FFS implementation, requiring two synchronous writes (one to copy the inode to its new home and one to update the directory from which it was moved). Externalized inodes are kept in a dynamically-growable, file-like structure that is similar to the IFILE in BSD-LFS [Seltzer93]. Some differences are that the externalized inode structure grows as needed but does not shrink and its blocks do not move once they have been allocated.

### File system recovery

One concern that re-arranging file system metadata raises is that of integrity in the face of system failures and media corruption. Regarding the first, we have had no difficulties constructing an off-line file system recovery program much like the UNIX `fsck` utility [McKusick94]. Although inodes are no longer at statically determined locations, they can all be found (assuming no media corruption) by following the directory hierarchy. We also do not believe that embedded inodes increase the time required to complete failure recovery, since reading the directories is required for checking link

counts anyway. In fact, embedded inodes reduce the effort involved with verifying the link counts, since extra state need not be kept (a valid embedded inode has a link count of one plus the number of internal links).

The one potential problem that embedding inodes introduces is that an unfortunate media corruption can cause all files below a corrupted directory to be lost. Even though they are uncorrupted, the single way to find them will have been destroyed. This is unacceptable when compared to the current scheme, which loses a maximum of 64 files or directories when an inode block is destroyed. All files that become disconnected from the name hierarchy due to such a loss can still be found. Fortunately, we can employ a simple solution to this problem — redundancy. By augmenting the directory map with the physical location of each directory's inode, we eliminate the loss of directories and files below a lost directory. Although the absolute amount of loss may be slightly higher (because the density of useful information is higher), it should be acceptable given the infrequency of post-factory media corruption.

### Simplifying integrity maintenance

Although the original goal of embedded inodes was to reduce the number of separate disk requests, a pleasant side effect is that we can also eliminate one of the sequencing constraints associated with metadata updates [Ganger94]. In particular, by eliminating the physical separation between a name and the corresponding inode, C-FFS exploits a disk drive characteristic to atomically update the pair. Most disks employ powerful error correcting codes on each sector, which has the effect of eliminating (with absurdly high probability) the possibility of only part of the sector being updated. So, by keeping the two items in the same sector, we can guarantee that they will be consistent with respect to each other. For file systems that use synchronous writes to ensure proper sequencing, this can result in a two-fold performance improvement [Ganger94]. For more aggressive implementations (e.g., [Hagmann87, Chutani92, Ganger95]), this reduces complexity and the amount of book-keeping required.

### Directory sizes

A potential down-side of embedded inodes is that the directory size can increase substantially. While making certain that an inode and its name remain in the same sector, three directory entries with embedded 128-byte inodes can be placed in each 512-byte sector. Fortunately, as shown in Figure 4, the number of directory entries is generally small. For example, 94% of all directories would require only one 8 KB block on our file

servers. For many of these, embedded inodes actually fit into space that was allocated anyway (the minimum unit of allocation is a 1 KB fragment). Still, there are a few directories with many entries (e.g., several of over 1000 and one of 9000). For a large directory, embedded inodes could greatly increase the amount of data that must be read from disk in order to scan just the names in a directory (e.g., when adding yet another directory entry). We are not particularly concerned about this, since bandwidth is what disks are good at. However, if experience teaches us that large directories are a significant problem, one option is to use the external inode “file” for the inodes of such directories.

### 3.2 Grouping Small Files

Like embedded inodes, small file grouping is conceptually quite simple. C-FFS simply places several small files adjacently on the disk and read/writes the entire group as a unit. The three main issues that arise are identifying the disk locations that make up a group, allocating disk locations appropriately, and caching the additional group items before they have been requested or identified. We discuss each of these below.

#### Identifying the group boundaries

To better exploit available disk bandwidth, C-FFS moves groups of blocks to and from the disk at once rather than individually. To do so, it must be possible to determine, for any given block, whether it is part of a group and, if so, what other disk locations are in the same group. C-FFS does this by augmenting each inode with two fields identifying the start and length of the group.<sup>4</sup> Because grouping is targeted for small files, we don't see any reason to allow an inode to point to data in more than one group. In fact, C-FFS currently allows only the first block (the only block for 79% of observed files) of any file other than a directory to be part of a group. Identifying the boundaries of the group is a simple matter of looking in the inode, which must already be present in order to identify the disk location of the desired block.

#### Allocation for groups

Before describing what is different about the allocation routines used to group small files, we want to stress what is not different. Placement of data for large files remains unchanged and should exploit clustering technology [Peacock88, McVoy91]. Directories are also placed as in current systems, by finding a cylinder group

with many free blocks. (The fast file system actually performs directory allocation based on the number of free inodes rather than the amount of free space. With embedded inodes, however, this becomes both difficult and of questionable value.)

The main change occurs in deciding where to place the first block (or fragment) of a file. The standard approach is to scan the freelist for a free block in the cylinder group that contains the inode, always starting from the beginning. As files come and go, the free regions towards the front of the cylinder group become fragmented. C-FFS, on the other hand, tries to incorporate the new file block (or fragment) into an existing group associated with the directory that names the file. This succeeds if there is free space within one of the groups (perhaps due to a previous file deletion or truncation) or if there is a free block outside of a group that can be incorporated without causing the group to exceed its maximum size (currently hard-coded to 64 KB in our prototype, which approximates the knees of the curves in Figure 2). To identify the groups associated with a particular directory, C-FFS exploits the fact that using embedded inodes gives us direct access to all of the relevant inodes (and thus their group information) by examining the directory's inode (which identifies the boundaries of the first group) and scanning the directory (ignoring any inodes for directories). If the new block extends an existing group, C-FFS again exploits embedded inodes to scan the directory and update the group information for other group members. If the new block cannot be added to an existing group, the conventional approach is used to allocate the block and a new group with one member is created.

One point worth stressing about explicit grouping is that it does not require all blocks within the boundaries of a group to belong to inodes in a particular directory. Although C-FFS tries to achieve this ideal, arbitrary files can allocate blocks that end up falling within the boundaries of an unrelated group. The decision to allow this greatly simplifies the management of group information (i.e., it is simply a hint that describes blocks that are hopefully related). Although this could result in sub-optimal behavior, we believe that it is appropriate given the fact that reading/writing a few extra disk blocks has a small incremental cost and given the complexity of alternative approaches.

Multi-block directories are the exception to both the allocation routine described above and to the rule that only the first block of a file can be included in a group. Because scanning the contents of a directory is a common operation, C-FFS tries to ensure that all of its blocks are part of the first group associated with the directory (i.e., the one identified by the directory's inode). Fortunately, even with embedded inodes, most directories

<sup>4</sup>Although we have added fields to support grouping in the C-FFS prototype, it would seem reasonable to overload two of the indirect block pointers instead and simply disable grouping for large files.

(e.g., 94 percent for our file servers) will be less than one block in size. If a subsequent directory block must be allocated and can not be incorporated into the directory's first group, C-FFS selects one of the non-directory blocks in the first group, moves it (i.e., copies it to a new disk location and changes the pointer in the inode), and gives its location to the new directory block. Once again, this operation exploits embedded inodes to find a member of the first group from which to steal a block. Unfortunately, moving the group member's block requires two synchronous writes (one to write the new block and one to update the corresponding inode). A better metadata integrity maintenance mechanism (e.g., write-ahead logging or soft updates) could eliminate these synchronous writes. When no blocks are available for stealing, C-FFS falls back to allocating a block via the conventional approach (with a preference for a location immediately after the first group).

### Cache functionality required

Grouping requires that the file block cache provide a few capabilities, most of which are not new. In particular, C-FFS requires the ability to cache and search for blocks whose higher level identities (i.e., file ID and offset) are not known. Therefore, our file cache is indexed by both disk address<sup>5</sup>, like the original UNIX buffer cache, and higher-level identities, like the SunOS integrated caching and virtual memory system [Gingell87, Moran87]. C-FFS uses physical identities to insert newly-read blocks of a group into the cache without back-translating to discover their file/offset identities. Instead, C-FFS inserts these blocks into the cache based on physical disk address and an invalid file/offset identity. When a cache miss occurs for a file/offset that is a member of any group, C-FFS searches the cache a second time, by physical disk address (since it might have been brought in by a previous grouped access). If the block is present under this identity, C-FFS changes the file/offset identity to its proper value and returns the block without an additional disk read.

The ability to search the cache by physical disk address is also necessary when initiating a disk request for an entire group. When reading a group, C-FFS prunes the extent read based on which blocks are already in the cache (it would be disastrous to replace a dirty block with one read from the disk). When writing a group, C-FFS prunes the extent written to include only those blocks that are actually present in the cache and dirty.

Given that C-FFS requires the cache to support

<sup>5</sup>By physical disk address, we mean the address given to the device driver when initiating a disk request, as opposed to more detailed information about how the cylinder, surface and rotational offset at which the data are stored inside the disk.

| Specification  | Value     |
|----------------|-----------|
| Capacity       | 1 GB      |
| Cylinders      | 2700      |
| Surfaces       | 9         |
| Sectors/Track  | 84        |
| Rotation Speed | 5400 RPM  |
| Head Switch    | N/A       |
| One-cyl Seek   | 1 ms      |
| Average Seek   | 9/10.5 ms |
| Maximum Seek   | 22 ms     |

Table 2: Characteristics of the Seagate ST31200 drive.

lookups based on disk address, C-FFS uses this mechanism (rather than clustering or grouping information) to identify additional dirty blocks to write out with any given block. With this scheme, the main benefit provided by grouping is to increase the frequency with which dirty blocks are physically adjacent on the disk. C-FFS uses scatter/gather support to deal with non-contiguity of blocks in the cache. Were scatter/gather support not present, C-FFS would rely on more complicated extent-based memory management to provide contiguous regions of physical memory for reading and writing of both small file groups and large file clusters.

## 4 Performance Evaluation

This section reports measurements of our C-FFS implementation, which show that it can dramatically improve performance. For small file activity (both reads and writes), we observe order of magnitude reductions in the number of disk requests and factor of 5–7 improvements in performance. We observe no negative effects for large file I/O. Preliminary measurements of application performance show improvements of 10–300 percent.

### 4.1 Experimental Apparatus

All experiments were performed on a PC with a 120 MHz Pentium processor and 32 MB of main memory. The disk on the system is a Seagate ST31200 (see Table 2). The disk driver, originally taken from NetBSD, supports scatter/gather I/O and uses a C-LOOK scheduling algorithm [Worthington94]. The disk prefetches sequential disk data into its on-board cache. During the experiments, there was no other activity on the system, and no virtual memory paging occurred. In all of our experiments, we forcefully write back all dirty blocks before considering the measurement complete. Therefore, our disk request counts include all blocks dirtied

by a particular application or micro-benchmark.

The disk drive used in the experiments is several years old and can deliver only one-third to one-half of the bandwidth available from a state-of-the-art disk (as seen in Table 1). As a result, we believe that the performance improvements shown for embedded inodes and explicit grouping are actually conservative estimates. These techniques depend on high bandwidth to deliver high performance for small files, while the conventional approach depends on disk access times (which have improved much less).

The C-FFS implementation evaluated here is the default file system for the Intel x86 version of the exokernel operating system [Engler95]. It includes most of the functionality expected from an FFS-like file system. Its major limitations are that it currently does not support prefetching or fragments (the units of allocation are 4 KB blocks). Prefetching is more relevant for large files than the small files our techniques address and should be independent of both embedded inodes and explicit grouping. Fragments, on the other hand, are very relevant and we expect that they would further increase the value of grouping, because the allocator would explicitly attempt to allocate fragments within the group rather than taking the first available fragment or trying to optimize for growth by allocating new fragment blocks.

We compare our C-FFS implementation to itself with embedded inodes and explicit grouping disabled. Although this may raise questions about how solid our baseline is, we are comfortable that it is reasonable. Comparisons of our restricted C-FFS (without embedded inodes or explicit grouping) to OpenBSD's FFS on the same hardware indicate that our baseline is actually as fast or faster for most file system operations (e.g., twice as fast for file creation and writing, equivalent for deletion and reading from disk, and much faster when the static file cache size of OpenBSD limits performance). The performance differences are partially due to the system structure, which links the file system code directly into the application (thereby avoiding many system calls), and partially due to the file system implementation (e.g., aggressively clustering dirty file blocks based on physical disk addresses rather than logical relationships).

For our micro-benchmark experiments, we wanted to recreate the non-adjacency of on-disk placements observed on our file servers. Our simplistic approach was to modify the allocation routines. Rather than allocating a new file's first block at the first free location within the cylinder group, we start looking for free space at other locations within the cylinder group. For half of the files in a directory, we start looking at the last direct block of the previous entry. For the other half, we start looking at a random location within the cylinder group. The

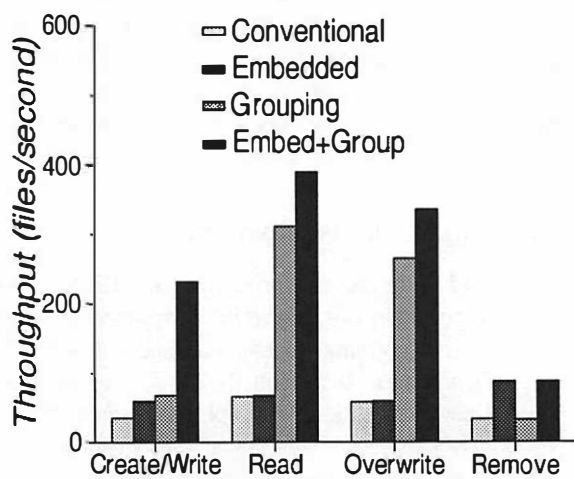
resulting data layouts are somewhat better than those observed on our file server, and therefore should favor the conventional layout scheme. We do not modify the inode allocation routines in any way, so inodes allocated in sequence for files in a given directory will tend to be packed into inode blocks much more densely than was observed on our server. Again, this favors the conventional organization. For the non-microbenchmark experiments, we do not do this.

## 4.2 Small File Performance

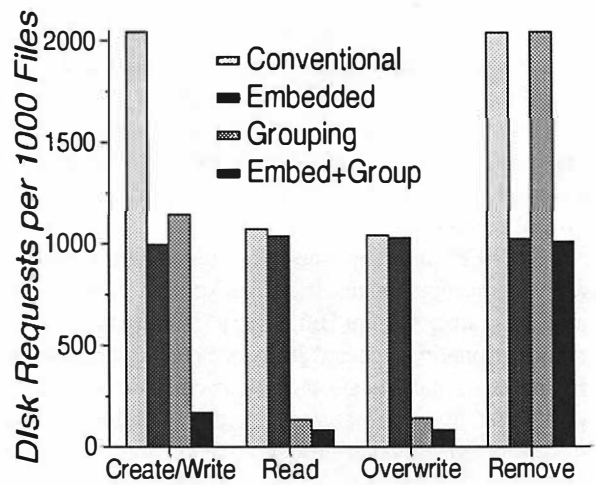
Figure 5 compares the throughput of 1 KB file operations supported by our prototype with embedded inodes, with explicit grouping, with neither and with both. The micro-benchmark, based on the small-file benchmark from [Rosenblum92], has four phases: create and write 10000 1KB files, read the same files in the same order, overwrite the same files in the same order, and then remove the files in the same order. To avoid the cost of name lookups in large directories, the files are spread among directories such that no single directory has more than 100 entries. In this comparison, both file systems are configured to use synchronous writes for metadata integrity maintenance (as is common among UNIX file systems).

For file creation, we observe a twentyfold reduction in the number of disk requests necessary when using both embedded inodes and explicit grouping. This results in a sevenfold increase in throughput. Half of the reduction in the number of disk requests comes from eliminating the synchronous writes required for integrity (by making the name and inode updates atomic) and half comes from writing the blocks for several new files with each disk request. It is interesting to note that half of the disk requests that remain are synchronous writes required because additional directory blocks are being forced into the directory's first group. This cost could be eliminated by a better integrity maintenance scheme (see below) or by not shuffling disk blocks in this way (which could involve a performance cost for later directory scanning operations). It is also interesting to observe that C-FFS with both embedded inodes and explicit grouping significantly outperforms C-FFS with either of these techniques alone (5 times fewer disk requests and 3–4 times the create/write throughput).

For file read and overwrite, we observe an order of magnitude reduction in the number of disk requests when using explicit grouping. This increases throughput by factors of 4.5–5.5. For these operations, embedded inodes alone provide marginal improvements. However, embedded inodes do provide measurable improvement when explicit grouping is in use, once again making the combination of the two the best option.

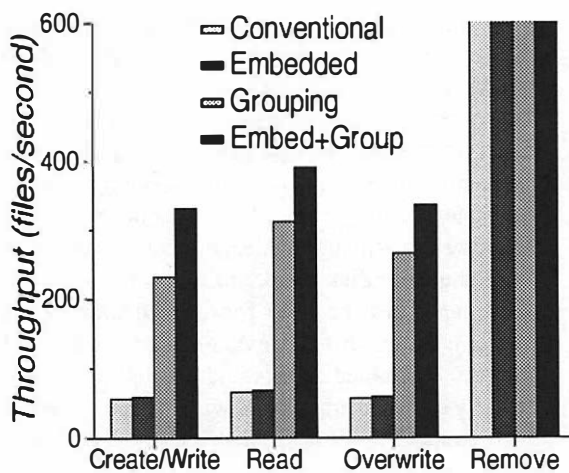


(a) Files per second

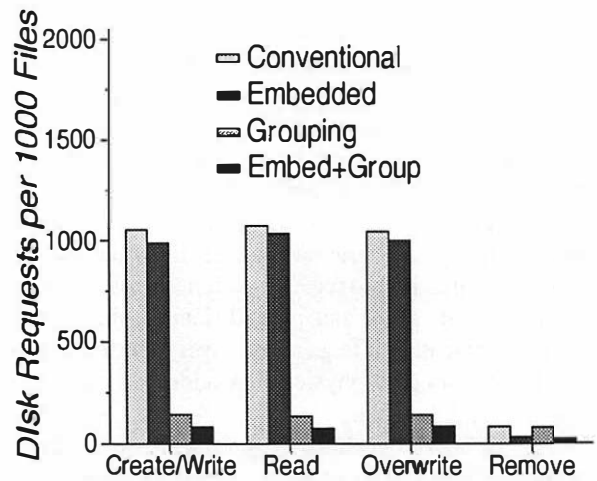


(b) Disk requests per 1000 files

Figure 5: Small file throughput when using synchronous writes for metadata integrity.



(a) Files/second



(b) Disk requests/1000 files

Figure 6: Small file throughput when using soft updates for metadata integrity. The “remove” throughput values are off the scale, ranging from 1500–2000 files per second.

For file deletion, we see a twofold decrease in the number of disk requests when using embedded inodes, since embedding the inodes eliminates one of the two sequencing requirements for file deletion. The second sequencing requirement, which relates to the relationship between the inode and the free map, remains. The result is a 250% increase in file deletion throughput, provided both by the reduction in the number of disk requests and improved locality (i.e., the same block gets overwritten repeatedly as the multiple inodes that it contains are re-initialized).

Because there exist techniques (e.g., soft updates [Ganger94]) that have been shown to effectively eliminate the performance cost of maintaining metadata integrity, we repeat the same experiments with this cost removed. Figure 6 shows these results. We have not yet actually implemented soft updates in C-FFS, but rather emulate it by using delayed writes for all metadata updates — [Ganger94] shows that this will accurately predict the performance impact of soft updates. The change in performance that we observe for the conventional file system is consistent with previous studies.

With the synchronous metadata writes removed, we still observe an order of magnitude reduction in disk requests when using explicit grouping for create/write, read and overwrite operations. The result is throughput increases of 4–7 times. Although the performance benefit of embedded inodes is lower for the create/write phase (because synchronous writes are not an issue), there is still a gain because embedded inodes significantly reduce the work involved with allocation for explicit grouping. As before, the combination of embedded inodes and explicit grouping provides the highest throughput for both the read and overwrite phases. With synchronous writes eliminated, file deletion throughput increases substantially. Although it has no interesting effect on performance (because resulting file deletion throughput is so high), embedding inodes halves the number of blocks actually dirtied when removing the files because there are no separate inode blocks.

### 4.3 File System Aging

To get a handle on the impact of file system fragmentation on the performance of C-FFS, we use an aging program similar to that described in [Herrin93]. The program simply creates and deletes a large number of files. The probability that the next operation performed is a file creation (rather than a deletion) is taken from a distribution centered around a desired file system utilization. After reaching the desired file system utilization for the first time, the aging program executes some number of additional file operations taken from the same distribution. The size of each file created is taken from the

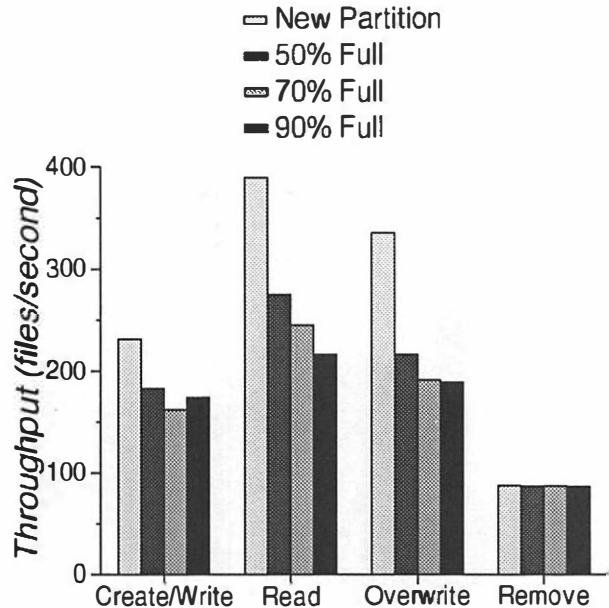


Figure 7: Small file throughput with embedded inodes and explicit grouping after aging the file system. Before running the small file micro-benchmark (without soft updates), the file system was aged by filling the file system to a desired capacity and then executing 100000 file create and delete operations. To increase the impact of the aging, we performed these experiments on a 128 MB partition. The four bars for each phase represent C-FFS performance for a fresh file system, for a 50%-full aged file system, for a 70%-full aged file system and for a 90%-full aged file system, respectively.

distribution measured on our file servers.

Figure 7 shows performance for the small file micro-benchmark after the file system has been aged. As expected, aging does have a significant negative impact on performance. At 70% capacity, the throughputs for the first three phases decrease by 30–40%. However, comparing these throughputs to those reported in the previous section, C-FFS still outperforms the conventional file system by a factor of 3–4. Further, our current C-FFS allocation algorithms do not reduce or compensate for fragmentation of free extents. We expect that the degradation due to file system aging can be significantly reduced by better allocation algorithms.

### 4.4 Large File Performance

Although C-FFS focuses on improving performance for small files, it is essential that it not reduce the performance that can be realized for large files. To verify that this is the case, we use a standard large file micro-

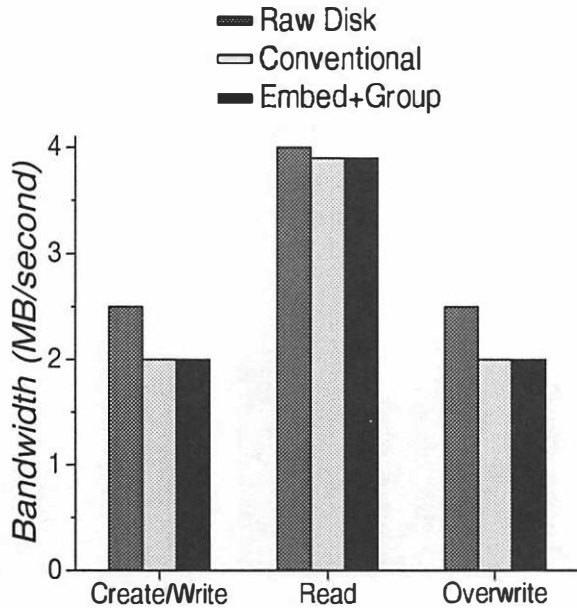


Figure 8: Large file bandwidth.

benchmark, which allocates and writes a large (32 MB) file sequentially, reads it sequentially and overwrites it sequentially. The results (shown in Figure 8) allow us to make two important points. First, using embedded inodes and explicit grouping has no significant effect on large file performance. Second, the C-FFS prototype, which supports clustering of large file data, delivers most of the disk's available bandwidth to applications for large files. (We believe that the somewhat disappointing file write bandwidth values are caused by software inefficiency; we have verified that it is not due to poor clustering.)

## 4.5 Applications

Figure 9 shows performance for four different applications which are intended to approximate some of the activities common to software development environments. As expected, we see significant improvements (e.g., a 50-66% reduction in execution time) for file-intensive applications like "pax" and "rm". For gmake, we see a much smaller improvement of only 10%. While such a small improvement could be viewed as a negative result, we were actually quite happy with it because of the extremely untuned nature of certain aspects of our system (which has just barely reached the point, at the time of this writing, where such applications can be run at all). In particular, process creation and shutdown (which are used extensively by the "gmake" application) are currently expensive. As a result, the time re-

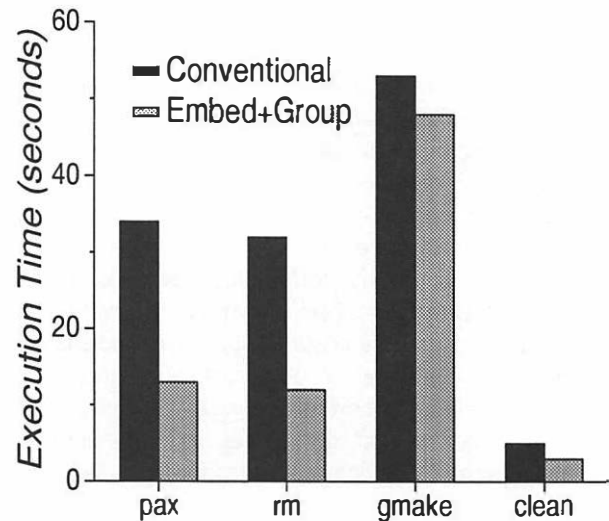


Figure 9: Application performance. "pax" uses the PAX utility to unpack a compressed archive file containing a substantial portion of the source tree for the default exokernel library operating system (about 1000 C files). "rm" uses the UNIX RM utility to recursively remove the directory tree created by "pax". "gmake" compiles a sub-directory containing 32 C files. "clean" removes the newly created object files from "gmake".

quired for "gmake" on our exokernel system is currently 250% greater than the corresponding execution time on OpenBSD. The same absolute improvement given a baseline equal to that of OpenBSD would represent a 25% reduction in execution time for this compute-intensive task.

We believe that, for most applications, the name space provides useful information about relationships between files that can be exploited by explicit grouping. However, there are some applications (e.g., the web page caches of many HTTP browsers) that explicitly randomize file accesses across several directories in order to reduce the number of files per directory. For workloads where the name space is a poor indicator of access locality, we expect grouping to reduce read performance slightly, because it reads from the disk more data than are necessary.

## 5 Related Work

Previous researchers and file system implementors have been very successful at extracting large fractions of

disks' maximum bandwidths for large files. One simple approach, which does have some drawbacks, is to increase the size of the basic file block [McKusick84]. Another is to cluster blocks (i.e., allocate them contiguously and read/write them as a unit when appropriate) [Peacock88, McVoy91]. Enumeration of a large file's disk blocks can also be significantly improved by using extents (instead of per-block pointers), B+ trees [Sweeney96] and/or sparse bitmaps [Herrin93]. We build on previous work by trying to exploit disk bandwidth for small files and metadata.

To increase the efficiency of the many small disk requests that characterize accesses to small files and metadata, file systems often try to localize logically related objects. For example, the Fast File System [McKusick84] breaks the file system's disk storage into cylinder groups and attempts to allocate most new objects in the same cylinder group as related objects (e.g., inodes in same cylinder group as containing directory and data blocks in same cylinder group as owning inode). Similarly, several researchers have investigated the value of moving the most popular (i.e., most heavily used) data to the centermost disk cylinders in order to reduce disk seek distances [Vongsathorn90, Ruemmler91, Staelin91]. As described in Section 2, simply locating related objects near each other offers some performance gains, but such locality affects only the seek time and is thus limited in scope. Co-locating related objects and reading/writing them as a unit offers qualitatively larger improvements in performance.

Immediate files are a form of co-location for small files. The idea, as proposed by [Mullender84], is to expand the inode to the size of a block and include the first part of the file in it. They found that over 60% of their files could be kept with the inode in a single block (and therefore both could be read or written with a single disk request). Another way to look at the same idea is simply that the inode is moved to the first block of the file. One potential down-side to this approach is that it replaces co-location of possibly related inodes (in an inode block) with co-location of an inode and its file data, which can reduce the performance of operations that examine file attributes but not file data. One simple application of the basic idea, however, is to put the data for very small files in the "normally" sized inode, perhaps replacing the space used by block pointers.

The log-structured file system's answer to the disk performance problem is to delay, remap and cluster all new data, only writing large chunks to the disk [Rosenblum92]. So long as neither cleaning nor read traffic represent significant portions of the workload, this will offer the highest performance. Unfortunately, while it may be feasible to limit cleaning activity to idle pe-

riods [Blackwell95], anecdotal evidence, measurements of real systems (e.g., [Baker91]), and simulation studies (e.g., [Dahlin94]) all suggest that main memory caches have not eliminated read traffic as hoped. Our work attempts to achieve performance improvements for both reads and writes of small files and metadata. While we are working within the context of a conventional update-in-place file system, we could easily see co-location being used in a log-structured file system to improve performance when reads become necessary.

One of the extra advantages of embedded inodes is the elimination of one sequencing constraint when creating and deleting files. There are several more direct and more comprehensive approaches to reducing the performance cost of maintaining metadata integrity, including write-ahead logging [Hagmann87, Chutani92, Journal92, Sweeney96], shadow-paging [Chamberlin81, Stonebraker87, Chao92, Seltzer93] and soft updates [Ganger95]. As shown in Section 4, our work complements such approaches.

Of course, there is a variety of other work that has improved file system performance via better caching, prefetching, write-back, indexing, scheduling and disk array mechanisms. We view our work as complementary to these.

## 6 Discussion

The C-FFS implementation described and evaluated in this paper is part of the experimental exokernel OS [Engler95]. We have found the exokernel to be an excellent platform for systems research of this kind. In particular, our first proof-of-concept prototype was extremely easy to build and test, because it did not have to deal with complex OS internals and it did not have to pay high overheads for being outside of the operating system.

However, the system-related design challenges for file systems in an exokernel OS (which focuses on distributing control of resources among applications) are different from those in a more conventional OS. An implementation of C-FFS for OpenBSD is underway, both to allow us to better understand how it interacts with an existing FFS and to allow us to more easily transfer it to existing systems. Early experience suggests that C-FFS changes some of the locking and buffer management assumptions made in OpenBSD, but there seem to be no fundamental roadblocks.

In this paper, we have compared C-FFS to an FFS-like file system, both with and without additional support for eliminating the cost of metadata integrity maintenance. Although we have not yet performed measurements, it is also interesting to compare C-FFS to other file systems

(in particular, the log-structured file system). We believe that C-FFS can match the write performance of LFS, so long as the name space correctly indicates logical relationships between files. For read performance, the comparison is more interesting. C-FFS will perform best when read access patterns correspond to relationships in the name space. LFS will perform best when read access patterns exactly match write access patterns. Although experiments is needed, we believe that C-FFS is likely to outperform LFS in many cases, especially when multiple applications are active concurrently.

In this paper, we investigate co-locating files based on the name space; other approaches based on application-specific knowledge are worth investigating. For example, one application-specific approach is to group files that make up a single hypertext document [Kaashoek96]. We are investigating extensions to the file system interface to allow this information to be passed to the file system. The result will be a file system that groups files based on application hints when they are available and name space relationships when they are not.

Our experience with allocation for small file grouping is preliminary, and there are a variety of open questions. For example, although the current C-FFS implementation allows only one block from a file to belong to a group, we suspect that performance will be enhanced and fragmentation will be reduced by allowing more than one. Also, C-FFS currently allows a group to be extended to include a new block even if it must also include an unrelated block that had (by misfortune) been allocated at the current group's boundary. We believe, based on our underlying assumption that reading an extra block incurs a small incremental cost, that this choice is appropriate. However, measurements that demonstrate this and indicate a maximum size for such holes are needed. Finally, allocation algorithms that reduce and compensate for the fragmentation of free space caused by aging will improve performance significantly (by 40–60%, according to the measurements in Section 4.3).

## 7 Conclusions

C-FFS combines embedded inodes and explicit grouping of files with traditional FFS techniques to obtain high performance for both small and large file I/O (both reads and writes). Measurements of our C-FFS implementation show that the new techniques reduce the number of disk requests by an order of magnitude for standard small file activity benchmarks. For the system under test, this translates into performance improvements of a factor of 5–7. The new techniques have no negative impact on large file I/O; the FFS clustering still delivers maximal performance. Preliminary experiments

with real applications show performance improvements of 10–300 percent.

## Acknowledgement

We thank Kirk McKusick and Bruce Worthington for early feedback on these ideas. We thank Garth Gibson, John Wilkes and the anonymous reviewers for detailed feedback in the later stages. We also thank the other members of the Parallel and Distributed Operating Systems research group at MIT, especially Héctor Briceño, Dawson Engler, Anthony Joseph, Eddie Kohler, David Mazières, Costa Sapuntzakis, and Josh Tauber for their comments, critiques and suggestions. In particular, we thank Costa who, while implementing C-FFS for OpenBSD, has induced many detailed discussions of design issues related to embedded inodes, explicit grouping and file systems in general. We especially thank Héctor and Tom Pinckney for their efforts as the final deadline approached.

## References

- [Baker91] M. Baker, J. Hartman, M. Kupfer, K. Shirriff, J. Ousterhout, "Measurements of a Distributed File System", *ACM Symposium on Operating Systems Principles*, 1991, pp. 198–212.
- [Blackwell95] T. Blackwell, J. Harris, M. Seltzer, "Heuristic Cleaning Algorithms in Log-Structured File Systems", *USENIX Technical Conference*, January 1995, pp. 277–288.
- [Chamberlin81] D. Chamberlin, M. Astrahan, et. al., "A History and Evaluation of System R", *Communications of the ACM*, Vol. 24, No. 10, 1981, pp. 632–646.
- [Chao92] C. Chao, R. English, D. Jacobson, A. Stepanov, J. Wilkes, "Mime: A High-Performance Parallel Storage Device with Strong Recovery Guarantees", Hewlett-Packard Laboratories Report, HPL-CSP-92-9, November 1992.
- [Chutani92] S. Chutani, O. Anderson, M. Kazar, B. Leverett, W. Mason, R. Sidebotham, "The Episode File System", *Winter USENIX Conference*, January 1992, pp. 43–60.
- [Dahlin94] M. Dahlin, R. Wang, T. Anderson, D. Patterson, "Cooperative Caching: Using Remote Client Memory to Improve File System Performance", *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, November 1994, pp. 267–280.
- [Engler95] D. Engler, M.F. Kaashoek, J. O'Toole Jr., "Exokernel: an operating system architecture for application-level resource management", *ACM Symposium on Operating Systems Principles*, Dec. 1995, pp. 251–266.
- [Forin94] A. Forin, G. Malan, "An MS-DOS File System for UNIX", *Winter USENIX Conference*, January 1994, pp. 337–354.

- [Ganger94] G. Ganger, Y. Patt, "Metadata Update Performance in File Systems", *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, November 1994, pp. 49–60.
- [Ganger95] G. Ganger, Y. Patt, "Soft Updates: A Solution to the Metadata Update Problem in File Systems", Technical Report CSE-TR-254-95, University of Michigan, Ann Arbor, August 1995.
- [Gingell87] R. Gingell, J. Moran, W. Shannon, "Virtual Memory Architecture in SunOS", *Summer USENIX Conference*, June 1987, pp. 81–94.
- [Hagmann87] R. Hagmann, "Reimplementing the Cedar File System Using Logging and Group Commit", *ACM Symposium on Operating Systems Principles*, November 1987, pp. 155–162.
- [Herrin93] E. Herrin, R. Finkel, "The Viva File System", Technical Report #225-93, University of Kentucky, Lexington, 1993.
- [HP91] Hewlett-Packard Company, "HP C2247 3.5-inch SCSI-2 Disk Drive – Technical Reference Manual", Edition 1, December 1991.
- [HP92] Hewlett-Packard Company, "HP C2244/45/46/47 3.5-inch SCSI-2 Disk Drive Technical Reference Manual", Part Number 5960-8346, Edition 3, September 1992.
- [HP96] Hewlett-Packard Company, <http://www.dmo.hp.com/disks/oemdisk/c3653a.html>, June 1996.
- [Journal92] NCR Corporation, "Journaling File System Administrator Guide, Release 2.00", NCR Document D1-2724-A, April 1992.
- [Kaashoek96] M.F. Kaaskoek, D. Engler, G. Ganger, D. Wallach, "Server Operating Systems", *ACM SIGOPS European Workshop*, September 1996, pp. 141–148.
- [McKusick84] M. McKusick, W. Joy, S. Leffler, R. Fabry, "A Fast File System for UNIX", *ACM Transactions on Computer Systems*, Vol. 2, No. 3, August 1984, pp. 181–197.
- [McKusick94] M. McKusick, T.J. Kowalski, "Fscck – The UNIX File System Check Program", *4.4 BSD System Manager's Manual*, O'Reilly & Associates, Inc., Sebastopol, CA, 1994, pp. 3:1–21.
- [McVoy91] L. McVoy, S. Kleiman, "Extent-like Performance from a UNIX File System", *Winter USENIX Conference*, January 1991, pp. 1–11.
- [Moran87] J. Moran, "SunOS Virtual Memory Implementation", *European UNIX Users Group (EUUG) Conference*, Spring 1988, pp. 285–300.
- [Mullender84] S. Mullender, A. Tanenbaum, "Immediate Files", *Software-Practice and Experience*, 14 (4), April 1984, pp. 365–368.
- [Ousterhout85] J. Ousterhout, H. Da Costa, D. Harrison, J. Kunze, M. Kupfer, J. Thompson, "A Trace-Driven Analysis of the UNIX 4.2 BSD File System", *ACM Symposium on Operating System Principles*, 1985, pp. 15–24.
- [Peacock88] J.K. Peacock, "The Counterpoint Fast File System", *USENIX Winter Conference*, February 1988, pp. 243–249.
- [Quantum96] Quantum Corporation, <http://www.quantum.com/products/atlas2/>, June 1996.
- [Rosenblum92] M. Rosenblum, J. Ousterhout, "The Design and Implementation of a Log-Structured File System", *ACM Transactions on Computer Systems*, 10 (1), February 1992, pp. 25–52.
- [Rosenblum95] M. Rosenblum, E. Bugnion, S. Herrod, E. Witchel, A. Gupta, "The Impact of Architectural Trends on Operating System Performance", *ACM Symposium on Operating Systems Principles*, Dec. 1995, pp. 285–298.
- [Ruemmler91] C. Ruemmler, J. Wilkes, "Disk Shuffling", Technical Report HPL-CSP-91-30, Hewlett-Packard Laboratories, October 3, 1991.
- [Ruemmler93] C. Ruemmler, J. Wilkes, "UNIX Disk Access Patterns", *Winter USENIX Conference*, January 1993, pp. 405–420.
- [Seagate96] Seagate Technology, Inc., <http://www.seagate.com/stor/storstop.shtml>, June 1996.
- [Seltzer93] M. Seltzer, K. Bostic, M. McKusick, C. Staelin, "An Implementation of a Log-Structured File System for UNIX", *Winter USENIX Conference*, January 1993, pp. 201–220.
- [Seltzer95] M. Seltzer, K. Smith, M. Balakrishnan, J. Chang, S. McMains, V. Padmanabhan, "File System Logging Verses Clustering: A Performance Comparison", *USENIX Technical Conference*, January 1995, pp. 249–264.
- [Smith96] K. Smith, M. Seltzer, "A Comparison of FFS Disk Allocation Policies", *USENIX Technical Conference*, January 1996, pp. 15–25.
- [Staelin91] "Smart Filesystems", *Winter USENIX Conference*, 1991, pp. 45–51.
- [Stonebraker81] M. Stonebraker, "Operating System Support for Database Management", *Communications of the ACM*, 24 (7), 1981, pp. 412–418.
- [Stonebraker87] M. Stonebraker, "The Design of the POSTGRES Storage System", *Very Large Data Base Conference*, September 1987, pp. 289–300.
- [Sweeney96] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, G. Peck, "Scalability in the XFS File System", *USENIX Technical Conference*, 1996, pp. 1–14.
- [Vongsathorn90] P. Vongsathorn, S. Carson, "A System for Adaptive Disk Rearrangement", *Software-Practice and Experience*, Vol. 20, No. 3, March 1990, pp. 225–242.
- [Worthington94] B. Worthington, G. Ganger, Y. Patt, "Scheduling Algorithms for Modern Disk Drives", *ACM SIGMETRICS Conference*, May 1994, pp. 241–251.
- [Worthington95] B. Worthington, G. Ganger, Y. Patt, J. Wilkes, "On-Line Extraction of SCSI Disk Drive Parameters", *ACM SIGMETRICS Conference*, May 1995, pp. 146–156.



# Observing the Effects of Multi-Zone Disks \*

Rodney Van Meter  
*Information Sciences Institute*  
*University of Southern California*

## Abstract

Current generations of hard disk drives use a technique known as **zoned constant angular velocity** (ZCAV), taking advantage of the geometry to increase total disk capacity by varying the number of disk sectors per track with the distance from the spindle. A side effect of this is that the transfer rate also varies with sector address. We analytically estimated and measured this effect on file system performance on a BSD Fast File System, showing a drop of roughly 25% in peak transfer rate depending on head position. We also show that, while ZCAV effects cannot be ignored, a simple linear model adequately estimates the performance from the few parameters normally available in disk drive spec sheets.

## 1 Introduction

Many magnetic disk drives use a technique known as *zoned constant angular velocity* (ZCAV), taking advantage of the geometry to increase total disk capacity by varying the number of disk sectors per track with the distance from the spindle. A side effect of this is that the transfer rate also varies with block address.

Despite some excellent recent work on modeling the behavior of disk drives [10, 14], the effects of ZCAV have generally not been taken into account in the design of file systems. Worthington et al [13] built a disk model which includes zone information, but the emphasis of their work is on disk scheduling algorithms to reduce latency, rather than improve throughput. Ghandeharizadeh has suggested [4] that file placement be adjusted based on access history to take advantage of ZCAV effects, but no work has measured the effects directly.

\*This research was sponsored by the Advanced Research Projects Agency under Contract No. DABT63-93-C-0062. Views and conclusions contained in this report are the authors' and should not be interpreted as representing the official opinion or policies, either expressed or implied, of ARPA, the U.S. Government, or any person or agency connected with them.

The Microsoft Tiger Video Server [2] uses a simple placement algorithm in which primary data is placed on outer tracks and secondary (redundant, infrequently-accessed) data is placed on inner tracks.

We estimated and measured the effect of ZCAV on file system performance on a BSD fast file system, showing a drop of roughly 25% in peak transfer rate depending on head position. We also show that, while ZCAV effects cannot be ignored, a simple linear model adequately estimates the performance from the few parameters normally available in disk drive spec sheets.

The rest of the paper is organized as follows. ZCAV is explained in detail in section 2. In section 3 we extract the zoning information for the disk drive used in our experimental analysis. In the following section an analytic model for estimating ZCAV drive performance is presented. Then, our experimental setup is described, followed by our measured results and conclusions.

## 2 Zoned Constant Angular Velocity

Magnetic disk drives consist of one or more rotating platters on a common spindle. Data is written and read by magnetic *heads*, generally one per surface (often with a spare surface, so that the number of heads is one less than twice the number of platters). A *track* is a concentric circle on one surface. The collection of tracks at the same distance from the spindle on each surface constitute a *cylinder*. A track consists of a number of *sectors* (occasionally called *blocks*), the smallest unit of data that can be read or written by the drive (typically 512 or 1024 bytes, but theoretically any number). The triple <cylinder, head, sector> uniquely defines a location on the drive. See [10, 13] for good introductions to disk architecture.

ZCAV is a technique adopted by hard disk manufacturers to increase the capacity of disk drives. Outer tracks, which are longer, contain more sec-

tors than the shorter inner tracks. The cylinders are grouped into *zones* that all have the same number of sectors per track. Some manufacturers refer to this as *Zoned Bit Recording, ZBR*. It is referred to as *notches* or a *notched drive* in the Small Computer Systems Interface (SCSI) specification [1].

As a side effect of this, since the time per rotation is constant, the number of sectors read per second (and hence the transfer rate) is higher on outer tracks. The read and write electronics must be able to keep up with the higher data rates required.

Compact disks (hence, CD-ROM) and old 400KB and 800KB Macintosh floppy drives achieved similar increases in density by varying the rotation speed to achieve *constant linear velocity*. For high-performance hard disk drives this is impractical, since each seek also means fighting high angular momentum to reach the correct speed, increasing the latency on seeks to an unacceptable level.

As table 1 shows<sup>1</sup>, the transfer rate of the outer zones of current disk drives from a major manufacturer exceeds that of the inner zones by factors ranging from 1.45 to 1.9. It is interesting to note that the disks with the highest capacity are not necessarily those with the highest ratio of inner to outer transfer rate.

The ST31200, for example, falls off from 47.2 to 26.8 Mbps, a drop of 43%. Thus, if the disk is operating mainly in the inner regions of the disk, performance can be expected to fall to just over half of the peak rate. Although not generally stated in the user manuals for the disk drives, empirical evidence indicates that the lower-numbered blocks (for a SCSI command set interface) are stored on the outer tracks.

Note that these transfer rates are internal preformat transfer rates; we will use this information to calculate the user data rate in the next section.

Manufacturers sometimes report an "average" number of sectors per track for ZCAV disk drives. This number appears to be arrived at by totalling the number of sectors in the drive and dividing by the number of tracks. It does not attempt to reflect the fact that a higher percentage of the sectors are in tracks with more sectors. This average is useful for filling in the BSD disk format information (see the manual pages for `fs` and `newfs`), which retains the cylinder, head, sector model.

<sup>1</sup>Most of these values were retrieved from Seagate's web site (<http://www.seagate.com>), but the availability of data there varies.

### 3 Determining Zone Information

SCSI is a commonly used interface for disk drives, and all of the drives we deal with in this paper have SCSI interfaces. At the SCSI command level, sectors are referred to by a logical block address, which the device controller maps to a physical location.

Some information about the disk geometry is often available through the **MODE SENSE Notch** and **Partition Page** on SCSI disk drives. This page reports the number of notches. The two drives used for this paper, the ST31200 and the ST11200, both report 23 notches in this page. On some drives it is possible to read some information about each zone using **MODE SELECT** and **MODE SENSE**. However, not all drives implement this functionality. The ST31200 supports this, but the ST11200 does not. The ST31200 only reports the number of cylinders in a zone, however, not the number of sectors per track or the total number of sectors in the zone.

More detailed information can be obtained by using **SEND DIAGNOSTIC** and **RECEIVE DIAGNOSTIC** with the **TRANSLATE ADDRESS** page. This provides the cylinder, head and sector number for each logical block, allowing easy determination of the number of sectors on a track, as well as two other important performance factors: the delay incurred by switching tracks and by switching cylinders, measured in sectors. It is interesting to note that on a Sparc 20/51 each address translation takes roughly 50 milliseconds, clearly at least one order of magnitude more than the actual translation requires. The reason for this delay is currently unknown.

The intratrack instantaneous transfer rate can be determined by multiplying the number of bytes per track by the revolutions per second,

$$\frac{\text{bytes}}{\text{track}} * \frac{\text{revs}}{\text{second}}$$

To find the sustained user rate for long transfers, this must be multiplied by the factor

$$\frac{h * s}{h * s + (h - 1) * g_t + g_c}$$

where  $h$  is the number of heads (tracks per cylinder),  $s$  is the sectors per track,  $g_t$  is the track-switch skew (gap) (measured in sectors) and  $g_c$  is the cylinder-switch skew (also in sectors). When reading continuously, the drive executes  $h - 1$  track switches plus one cylinder switch, per cylinder read.

Table 2 gives the detailed zone information for the Seagate ST11200 used in these experi-

| Drive             | capacity<br>(GB) | min internal<br>xfer rate<br>(Mbps) | max internal<br>xfer rate<br>(Mbps) | ratio |
|-------------------|------------------|-------------------------------------|-------------------------------------|-------|
| Barracuda ST11950 | 1.69             | 34.3                                | 56.5                                | 1.65  |
| Barracuda ST32171 | 2.25             | 75                                  | 120                                 | 1.60  |
| Elite ST43400     | 2.9              | 35                                  | 52                                  | 1.49  |
| Decathlon S5850A  | 0.71             | 32.45                               | 61.65                               | 1.90  |
| Hawk 4 ST15230    | 4.29             | 34                                  | 61                                  | 1.79  |
| Hawk 2XL ST31051  | 1.05             | 44                                  | 66                                  | 1.50  |
| Elite ST410800    | 9.09             | 44                                  | 65                                  | 1.47  |
| ST31200           | 1.06             | 26.8                                | 47.2                                | 1.76  |
| ST11200           | 1.05             | 23.2                                | 40.6                                | 1.75  |

Table 1: Transfer Rates for a Variety of Seagate Disks

| zone | start | cyls | heads | sec/trk | zonesec | totsec  | MB/sec. | trotgap | crotgap | adjMBs |
|------|-------|------|-------|---------|---------|---------|---------|---------|---------|--------|
| 1    | 0     | 205  | 15    | 94      | 289050  | 289050  | 4.34    | 18      | 28      | 3.62   |
| 2    | 205   | 30   | 15    | 93      | 41850   | 330900  | 4.29    | 17      | 28      | 3.61   |
| 3    | 235   | 93   | 15    | 92      | 128340  | 459240  | 4.25    | 17      | 27      | 3.56   |
| 4    | 328   | 33   | 15    | 91      | 45045   | 504285  | 4.20    | 17      | 27      | 3.52   |
| 5    | 361   | 68   | 15    | 88      | 89760   | 594045  | 4.06    | 17      | 26      | 3.39   |
| 6    | 429   | 144  | 15    | 84      | 181440  | 775485  | 3.88    | 16      | 25      | 3.24   |
| 7    | 573   | 38   | 15    | 83      | 47310   | 822795  | 3.83    | 16      | 25      | 3.19   |
| 8    | 611   | 78   | 15    | 80      | 93600   | 916395  | 3.69    | 15      | 24      | 3.09   |
| 9    | 689   | 79   | 15    | 77      | 91245   | 1007640 | 3.56    | 15      | 23      | 2.96   |
| 10   | 768   | 120  | 15    | 76      | 136800  | 1144440 | 3.51    | 15      | 23      | 2.91   |
| 11   | 888   | 81   | 15    | 75      | 91125   | 1235565 | 3.46    | 14      | 22      | 2.90   |
| 12   | 969   | 41   | 15    | 74      | 45510   | 1281075 | 3.42    | 14      | 22      | 2.86   |
| 13   | 1010  | 80   | 15    | 73      | 87600   | 1368675 | 3.37    | 14      | 22      | 2.81   |
| 14   | 1090  | 79   | 15    | 71      | 84135   | 1452810 | 3.28    | 14      | 21      | 2.72   |
| 15   | 1169  | 157  | 15    | 65      | 153075  | 1605885 | 3.00    | 13      | 20      | 2.49   |
| 16   | 1326  | 178  | 15    | 62      | 165540  | 1771425 | 2.86    | 12      | 19      | 2.38   |
| 17   | 1504  | 35   | 15    | 61      | 32025   | 1803450 | 2.82    | 12      | 19      | 2.34   |
| 18   | 1539  | 168  | 15    | 57      | 143640  | 1947090 | 2.63    | 11      | 18      | 2.19   |
| 19   | 1707  | 82   | 15    | 56      | 68880   | 2015970 | 2.59    | 11      | 17      | 2.15   |
| 20   | 1789  | 80   | 15    | 54      | 64800   | 2080770 | 2.49    | 11      | 17      | 2.06   |

Table 2: Extracted Zone Information for ST11200 with Calculated Transfer Rates

ments. This data was obtained by a modified version of John DiMarco's `scsiinfo`, using a `SEND DIAGNOSTIC/RECEIVE DIAGNOSTIC RESULTS` command pair with the `TRANSLATE ADDRESS` page for each block on the disk, then hand-extracting the zone boundaries. On disks that also support setting the active notch on the `MODE SELECT Notch and Partition Page`, it is possible to more directly extract the cylinder boundaries. The ST31200, for example, returns the notch size in cylinders, but not the total sectors in the notch. Determining the notch boundaries is also complicated by the track skew, sparing of sectors, and sector remaps. The capacity of each zone as listed does not take into account remapped or skipped sectors.

In table 2, the first column is zone number, starting from the outer edge, in accordance with block numbering. *start* is the cylinder number for the start of the zone. *cyls* is the number of cylinders in the zone. *heads*, the number of data heads used, is constant for the whole disk drive. *sec/trk* is the number of sectors per track in the zone. *zonesec*, the total number of sectors in the zone, is the product of the prior three columns; *totsec* is a running total of the *zonesec* column. The columns in the table labeled *trotgap* and *crotgap* are the track and cylinder skew. *MB/sec.* is the intratrack transfer rate determined as above, and *adjMBs* is the rate adjusted by the track and cylinder skew, as above. As the table shows, the reduction in transfer rate caused by fewer sectors in a zone can sometimes be almost completely offset by a reduction in the track skew. Compared with the values of 23.2 to 40.6 Mbps internal transfer rates cited in the manufacturer's manual, the adjusted values are 29% lower, and represent reasonable "not to exceed" values for system transfer rates. It is also worth noting that the drive reports 23 notches on the notch and partition page, but only twenty were discernable from the logical to physical block map. The transfer rate is graphed in figure 1, which is explained in detail in section 4.

## 4 Analytic Approach to Estimating Performance

In this section, we consider three abstract examples, then analyze the disk drive used for the experiments. Transfer rates here are quoted in sectors per revolution; multiplying by revolutions per second and bytes per sector (both constants) would give bytes/second.

The first example is a hypothetical three-zoned disk drive. The outer zone is 100 tracks of 175 sec-

tors, the middle zone is 100 tracks of 137 sectors, and the inner zone is 100 tracks of 100 sectors. This is overly simplistic but the ratios are common. The total capacity of the drive is  $100 * 175 + 100 * 137 + 100 * 100 = 41200$  sectors. Roughly 42% of the sectors are in the outer zone, 33% in the middle zone, and 24% in the inner zone. Figure 2 shows the transfer rate in each zone versus track number. Figure 3 plots the transfer rate versus block number. Note the different position of the boundary between zones relative to figure 2, due to the higher capacity of the outer zones.

The "average" number of sectors per track is 137. If we assume that each sector is accessed with equal frequency, the "average" transfer rate is  $(17500 * 175 + 13700 * 137 + 10000 * 100) / 41200$ , or 144 sectors/revolution, due to the higher probability of being in a high-sectors-per-track zone. This effect alone leads to an error of 5% when estimating performance based solely on the mean number of sectors per track.

As a second example, consider a more finely-grained zoning. Let the disk drive consist of one track of 175 sectors, one of 174 sectors, etc. down to an inner track of 100 sectors, for a total capacity  $C$  of

$$C = \sum_{i \in Z} s_i * t_i$$

where  $s_i$  is the number of sectors per track in zone  $i$ ,  $t_i$  is the number of tracks in the zone, and  $Z$  is the set of zones. In this case,  $s_i = 175 - i$  and  $t_i = 1$ , so this reduces to

$$C = \sum_{i=100}^{175} i = 10,450$$

sectors. The mean number of sectors per track is  $10,450 / 76 = 137.5$

Figure 4 shows transfer rate versus track number. Figure 5 shows the transfer rate versus block address for this example. Visually, it is nearly linear. A small  $n^2$  factor would be expected to cause the transfer rate to fall off more quickly at higher block numbers (fewer sectors per track mean fewer sectors per zone, meaning the advance to yet-smaller zones accelerates), as shown in figure 5. However, this factor appears to be unimportant, to first order.

The median transfer rate  $R_{med}$  is the transfer rate of block number  $C/2$ . In this case, block 5225 is on track 143, so it has a transfer rate of 143, 4% higher than the mean sectors per track.

The "average" transfer rate, again assuming equal probability of access for each sector, would be the

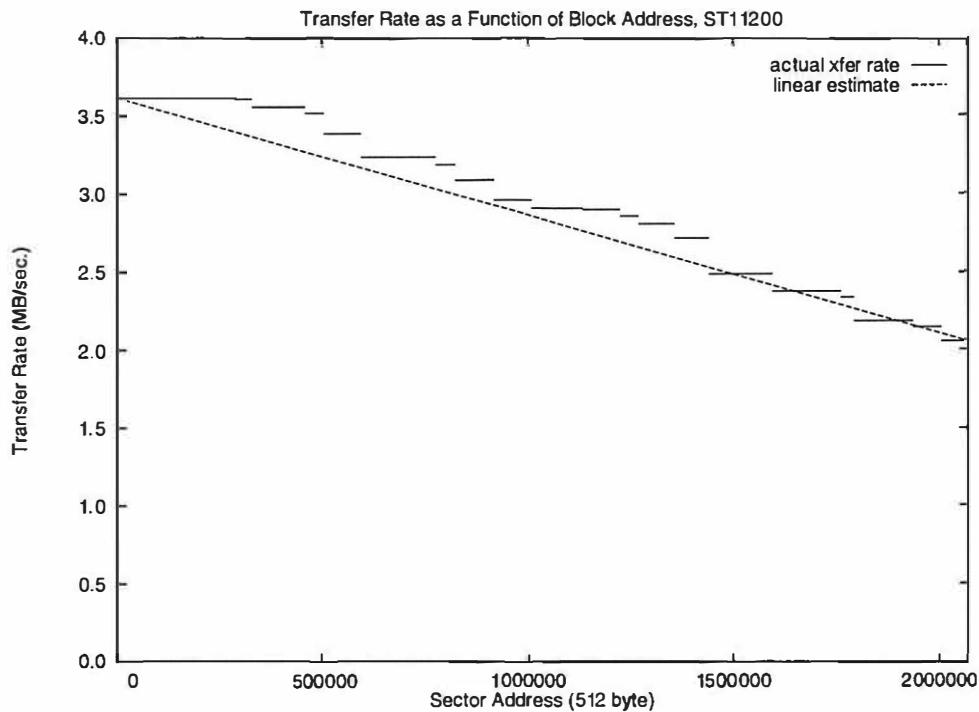


Figure 1: ST 11200 Zones with Calculated Transfer Rates

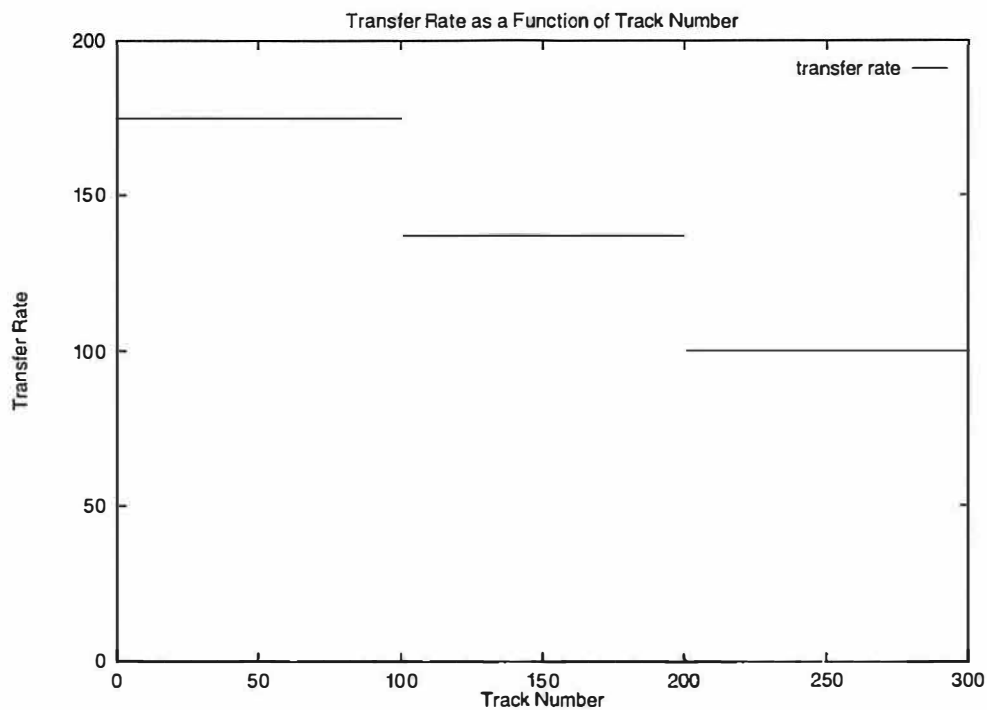


Figure 2: Three large zones, transfer rate v. track no.

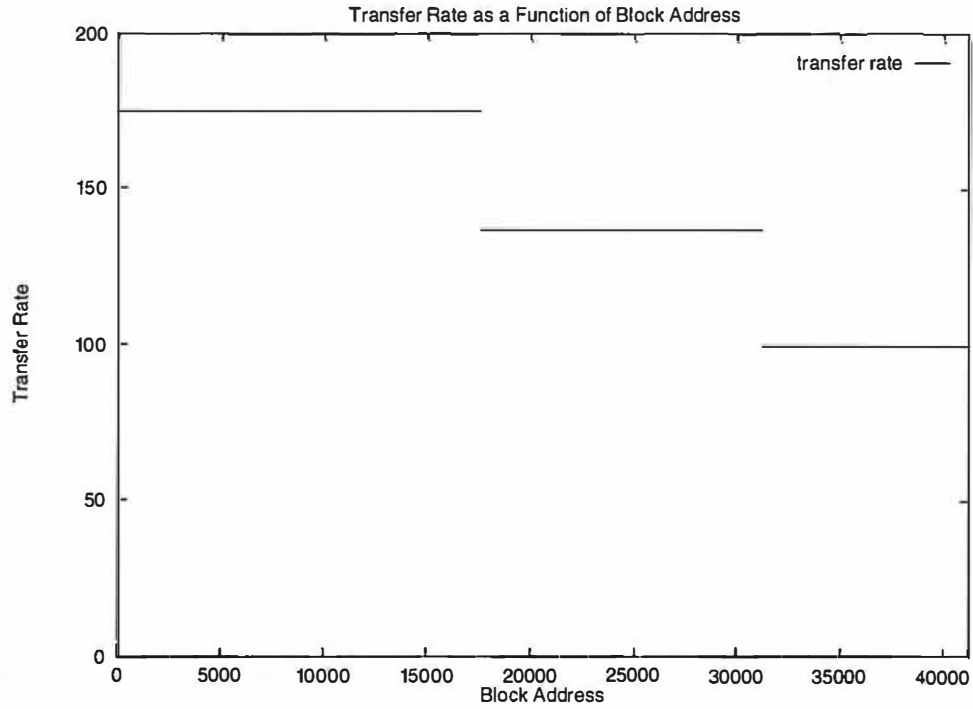


Figure 3: Three large zones, transfer rate v. block no.

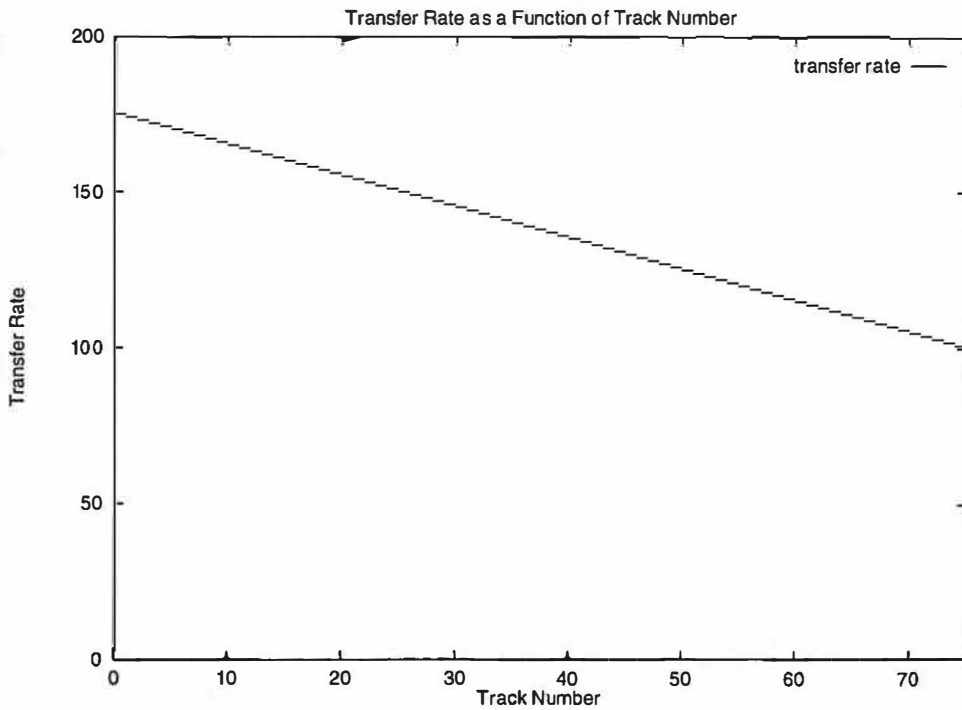


Figure 4: Single-track Zones, transfer rate v. track no.

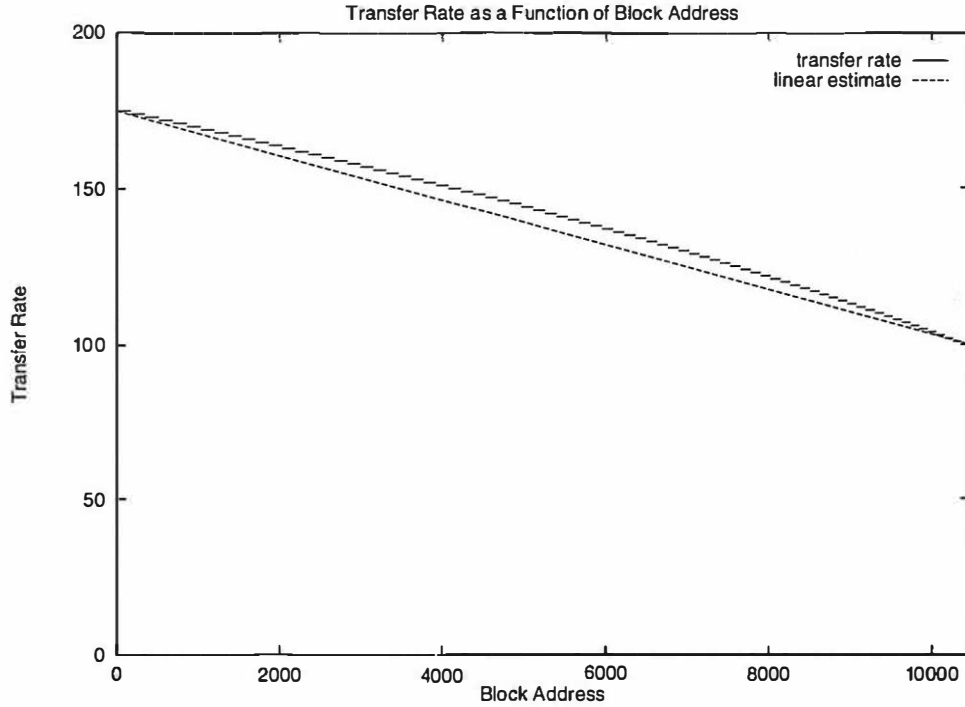


Figure 5: Single-track Zones, transfer rate v. block no.

sum of the transfer rates for each of the individual blocks, divided by the total number of blocks:

$$R_{ba} = \frac{\sum_{i \in Z} s_i^2 * t_i}{C}$$

$$= \frac{\sum_{i \in Z} s_i^2 * t_i}{\sum_{i \in Z} s_i * t_i}$$

This will *not* be equal to the “average” sectors per track reported by the manufacturer, times the rotations per second:

$$\frac{\text{avg.bytes}}{\text{second}} = \frac{\text{bytes}}{\text{sector}} * \frac{\text{avg.sectors}}{\text{rotation}} * \frac{\text{rotations}}{\text{second}}$$

In our example 2,  $R_{ba}$  simplifies to  $\sum_{i=100}^{175} i/C = (1,801,800 - 328,350)/C = 1,473,450/10,450 = 141$ , a very modest 2.5% increase from simply assuming it to be the mean of the max and min transfer rates. For all practical purposes, therefore, we can estimate the mean transfer rate as the mean of min and max, when the zoning is fine-grained and roughly linear with track number.

Figure 6 shows transfer rate versus block address for the Seagate ST31200, calculated based on the extracted zone information. It clearly shows the effects of more of the blocks being in the outer zones. The median transfer rate is 3.6 MB/sec, 10% higher than

the 3.25 arrived at by averaging the max and min rates. The average transfer rate, assuming each sector has equal probability of being accessed, is 3.47, still 6% higher than 3.25. Again, the curve varies only slightly from linear.

Figure 1 shows the transfer rate plotted against sector address for the ST11200 used for these experiments. A simple linear estimate is also plotted, running from the transfer rate at the outermost zone to the innermost zone. This shows a rough fit, with the maximum error from the true rate being approximately 8%. Thus, while far from perfect, this exceedingly simple model is significantly more accurate than assuming a fixed transfer rate, which may vary by 40%. In addition, this can be easily estimated from the data sheets typically supplied with disk drives.

A recommended first-order estimate of transfer rate, simple enough to be implemented in a guaranteed-I/O-rate file system, would therefore be

$$R(x) = 0.7r_{max} - \frac{0.7(r_{max} - r_{min})}{C} * x \quad (1)$$

where  $C$  is the disk capacity and  $r_{max}$  and  $r_{min}$  are the maximum and minimum internal transfer rates reported by the disk drive manufacturer. The factor 0.7 comes from our observation in section 3

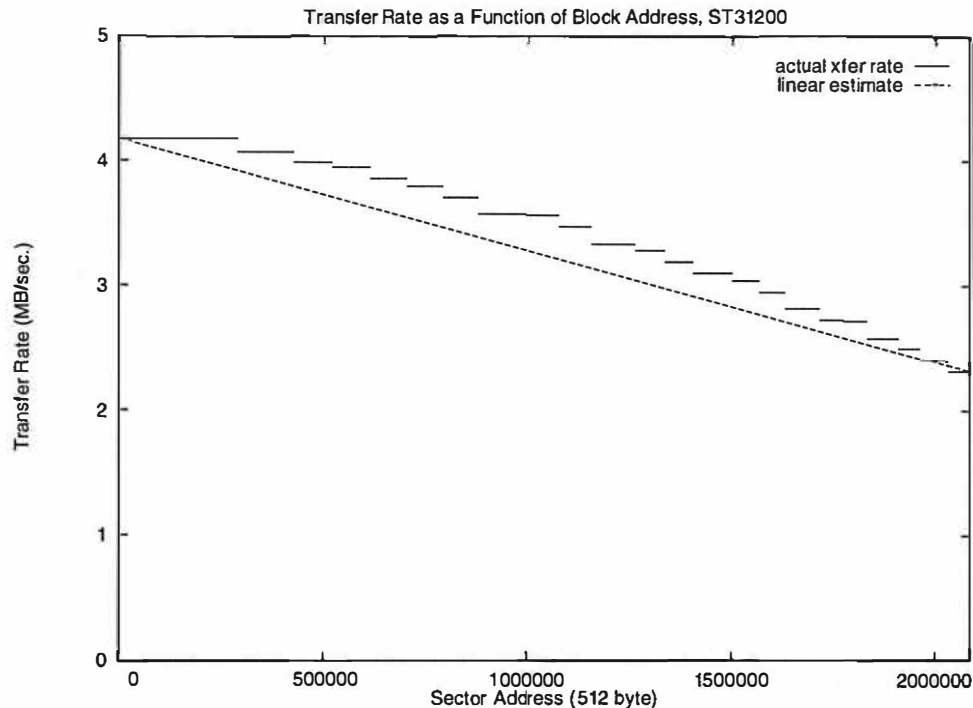


Figure 6: ST 31200 Zones with Calculated Transfer Rates

that transfer rates adjusted for sector overhead, error correction and track and cylinder skew results in a drop of approximately 29% from the manufacturer's listed transfer rates, which are instantaneous bit rates at the read/write head. Because this data is readily available, this factor can be incorporated quickly and easily by file system and device driver designers, without the necessity of tediously testing each possible disk drive. This transfer rate, of course, must be adjusted by the system's ability to sustain the I/O rate; as shown above, for a Sparc 10 running SunOS and a FFS, reads can run at device speeds, while writes run at approximately 80% of theoretical.

## 5 Experiment

### 5.1 Experimental Setup

These experiments were conducted on a Sparcstation 10 with 64 MB of main memory, and a 1.05 GB Seagate ST11200N disk drive. The actual bandwidth of this disk drive, as shown in table 2, varies from approximately 2.06 to 3.62 MB/sec., a factor of 1.75. Any read or write rate that exceeds that has clearly been the beneficiary of caching, either the file system's buffer cache or the disk drive's data block cache. According to the manual [11], this disk drive

has 23 zones, or notches, and an average (mean) of 73 sectors per track, 15 heads, 1,872 cylinders for a total of 28,080 tracks. The drive rotates at 5,411 rpm. Write caching at the disk is disabled; all writes are synchronous.

The file system is a SunOS UFS, essentially a BSD fast file system [5]. The partition used for these experiments begins at sector number 655,200 and extends 687MB to the end of the disk, as reported by `dkinfo`. Thus, according to table 2, the partition starts at a transfer rate of 3.24 MB/sec. and falls to 2.06, a drop of 36%. Unfortunately, due to hardware and disk partitioning limitations, it was not possible at the time this experiment was conducted to cover the entire span of a disk.

The basic experiment runs a loop that executes a modified version of Tim Bray's `bonnie` to write a 100MB file, unmount the partition (to clear the cache and commit all modified metadata), then read the file back. Then the script records the Bonnie data file layout, deletes the file, writes a 10MB file to the system, and repeats. Thus, we have the results for 100MB written at 10MB intervals. The free space falls from approximately 610MB (user space available) to 105MB in 50 steps.

## 5.2 Experimental Data

When measuring the effects of the ZCAV layout on file system performance, care must be taken as numerous other factors can contribute to changes in performance. They include:

- distance from metadata (increased seek times)
- free space fragmentation
- CPU performance and system loading
- buffer cache page replacement performance

Of course, the effect on performance will vary dramatically with the file system structure, which is generally operating-system specific; this is covered in the following section.

Our data shows that the write rate varies by a factor of 1.33 (2528 KB/sec. v. 1900 KB/sec., a drop of 25%) depending on head position, even over the limited range of our experiments. Evaluating the reads is more difficult due to the high variability, but if we choose the means from the same data runs as the writes, we see a 23% drop, 3295 KB/sec. v. 2547 KB/sec. at, respectively, 608 and 270 MB free.

Figure 7 shows the mean of ten runs<sup>2</sup> of our 100/10 benchmark. The error bars are 90% confidence intervals. Writes are also plotted with error bars, but they are too small to see at many data points. The results clearly show a drop in performance as the disk fills, until with about 280MB free space the curve takes a sharp, unexpected upward turn.

The write values are lower than the theoretical maximum due to inevitable missed rotations. Since the disk is not allowed to cache write data, typically at least one rotation must be missed at the end of each write request. Additionally, occasionally the file system writes some metadata to the drive, requiring a seek and write, with ensuing missed rotations. The measured values are fairly consistently approximately 80% of the calculated values, indicating approximately one missed rotation in five.

Examining the layout for the data files created by the Bonnie benchmark (examined using a modified version of Keith Smith's `fsblks` utility), as shown in figure 8, confirms the hypothesis that transfer rate is related to head position, as well as providing an explanation for the upward turn near the right-hand edge. The 10 MB filler files are getting laid down

<sup>2</sup>The runs actually used for these calculations are numbers 6 through 15; the first five represented progressive refinements of the measurement code and are discarded.

with holes between them which go unused until the disk nears full.

Returning to figure 7, the points labeled *calc* are calculated from run number 9, estimating the performance by integrating the transfer rate at each block in the file, using the transfer rates calculated for each zone in section 3. It clearly shows the same features (dips and peaks) as the write and read curves. The slight difference (approximately 5%) between the read curve and the calculated estimate is because the calculated estimate does not take into account real-world overheads for command processing and latency, and CPU time in the kernel and user process. This difference (for both read and write) will be system dependent and will have to be determined empirically.

The points labeled *lin-est* are calculated using the linear estimate shown in equation 1. The largest difference from the more correctly calculated values is 7%. This error is significant, but the simplicity of this linear estimate (both in ease of determination and ease of use) may make it an acceptable substitute for detailed zone calculations. The apparent better agreement of the linear estimate than the more realistic calculation above is coincidence; the linear estimate slightly underestimates performance compared to the non-linear effects of block address and geometry as described in section 4. Note that near the disk spindle (where the curve in figure 7 dips at 280,000 KB free), the agreement between the block calculation and linear estimate is better, as we would expect.

Reviewing our concerns expressed at the top of this section, our data has good repeatability, especially on writes. The buffer cache issue has been addressed by clearing the cache via remounting the partition. Free space fragmentation proved to not be a problem. The CPU and other system components appear to be up to the task of fully utilizing the disk, clearly showing the ZCAV effects we expected.

## 6 Conclusions

### 6.1 Dependence on File System Structure

One of the interesting aspects of this work is how repeatable the data proved to be, especially for writes. This clearly demonstrated that the SunOS file allocation code depends on the current state only; the recent history of file creations and deletions does not alter future file system allocation decisions. Note that this does not mean that disk fragmentation is

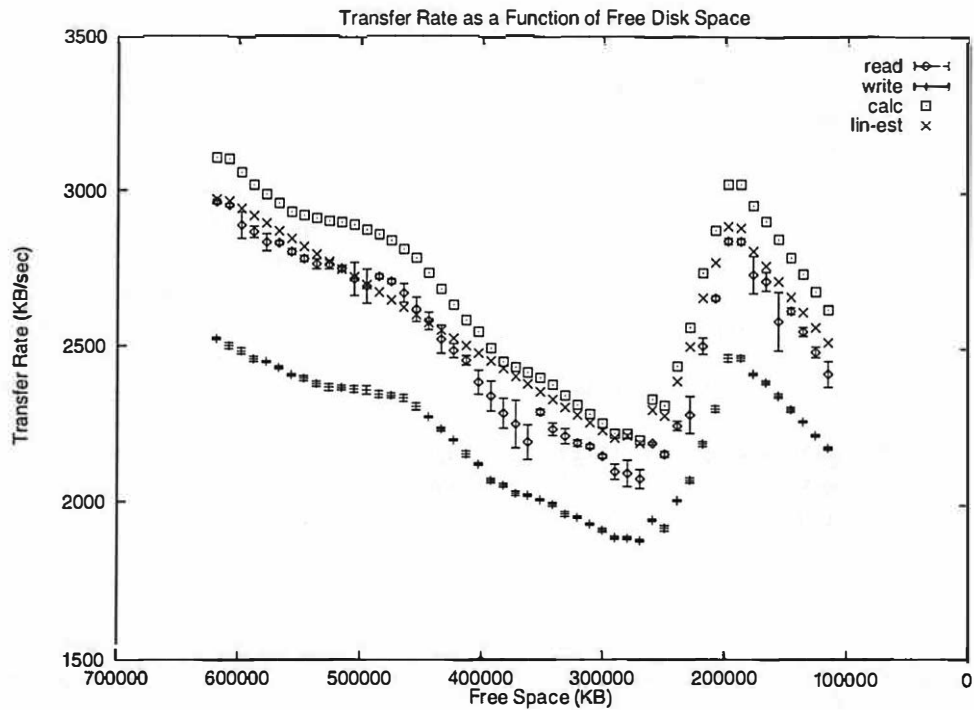


Figure 7: 10 runs of 100/10 ZCAV benchmark

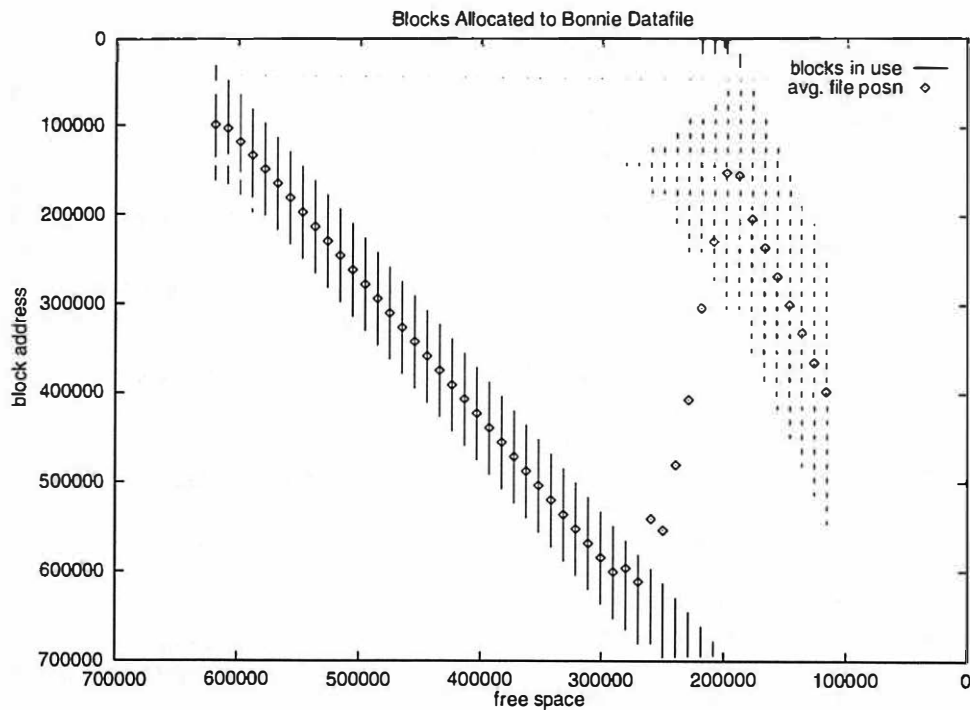


Figure 8: File layout for run # 7

not a general problem, only that once large areas of disk have been cleared of files, the reuse of that area is optimal (or at least predictable).

McVoy showed that a UFS can achieve good write performance [6]<sup>3</sup>. The file blocks are allocated contiguously and I/Os are performed in clusters, much like an extent-based file system. Our work benefits from this work.

Other possible file system structures, such as SGI's XFS [12], may depend on more dynamic, and hence complex, data structures, and may therefore not allocate blocks as predictably. A log-based file system [9] or disk device [3] clearly will not, in their present forms, allocate blocks in a fashion amenable to improving throughput by careful choice of blocks.

## 6.2 Impact on File System Allocation Policies

As proposed by Ghandeharizadeh [4], the idea of including a measure of ZCAV effects into a dynamic file relocater is appealing. Such functionality could be included in a file system defragmenter, moving older, less-frequently-accessed files to lower-transfer-rate areas of the disk.

It is clear that this effect needs to be taken into account for multimedia file systems and file systems (such as SGI's XFS [12] or Rangan's multimedia ropes [8]) that provide guaranteed throughput. However, to date these have all assumed disk bandwidth is fixed, rather than a function of block address.

Larger files accessed in large chunks, for which transfer rate is likely to be more important, should be allocated to blocks at the outer edges (for a Seagate SCSI drive, the lower-numbered blocks). Small files obviously do not need to be placed in a high-transfer rate location, as their transfer time will be dominated by latency. Large files accessed in small I/O requests also will not take good advantage of the transfer rate. Determining which files will take advantage of this may require cooperation from applications, perhaps via some form of hints [7].

Incorporating knowledge of the drive's ZCAV nature into the cleaner for a log-structured file system may be useful. Data should be packed toward the spindle, so that the open area for upcoming log writes will get to use the outer, faster regions of the disk. This could be expected to improve the write performance of the LFS by 25% or more, at the ex-

pense of slower reads, in keeping with the LFS philosophy.

## 6.3 Future Work

Obviously, we would like to try these experiments on a wider range of hardware and software platforms, especially different file systems. However, the point that performance varies with head position appears to have been adequately demonstrated. In particular, our work should be repeated with different families of disk drives from different manufacturers, to confirm both the hypothesis that ZCAV effects are user-visible, and the effectiveness of our proposed linear estimate.

Ideally, a publicly-available bank of information on drive types and zone information should be created. As more drive developers adopt standard methods of determining the zone information, of course, determining this information at boot time or file system configuration time becomes more feasible.

## 6.4 Conclusions

We have explained the underlying motivations behind ZCAV disk drives, and demonstrated that it does have an effect on file system performance for a BSD FFS. We have shown that it is possible, though somewhat tedious, to extract this information from at least some disk drives directly. We have proposed that a simple linear model relating transfer rate to block address should be adequate for most purposes. The measurable effect, at 23-25%, is less than the physical difference of 36% between the inner and outer disk edges of the tested partition, but still too large to ignore in performance-critical applications.

## Acknowledgments

This work would have been substantially more tedious if not for the generosity of Keith Smith (`fsblks`), John DiMarco (`scsiinfo`) and Tim Bray (`bonnie`) in making their code publicly available. John Heidemann, Ted Faber and Greg Finn provided useful technical and editorial suggestions. The anonymous referees and my shepherd, Bill Bolosky, improved the paper through their comments as well.

<sup>3</sup>McVoy noted, in fact, that exposing the drive's variable geometry to the system will complicate block allocation, especially in an extent-base FS.

## References

- [1] ANSI. *Information Technology - Small Computer Systems Interface - 2 (X3T9.2 rev 101)*, Sept. 1993.
- [2] W. J. Bolosky et al. The tiger video fileserver. In *Proc. Sixth International Workshop on Network and Operating System Support for Digital Audio and Video*, Apr. 1996. available at <ftp://ftp.research.microsoft.com/pub/tech-reports/Winter95-96/TR-96-09.ps>.
- [3] W. de Jonge, M. F. Kaashoek, and W. C. Hsieh. The logical disk: A new approach to improving file systems. In *Proc. Fourteenth ACM Symposium on Operating Systems Principles*, pages 15-28, Dec. 1993.
- [4] S. Ghandeharizadeh, D. J. Ierardi, D. Kim, and R. Zimmerman. Placement of data in multi-zone disk drives. rcvd from author, June 1995.
- [5] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 1989.
- [6] L. W. McVoy and S. R. Kleiman. Extent-like performance from a UNIX file system. In *Proc. 1991 USENIX Winter Technical Conference*, pages 33-43. USENIX, 1991.
- [7] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proc. 15th Annual ACM Symposium on Operating Systems Principles*, pages 79-95. ACM, Dec. 1995.
- [8] P. V. Rangan and H. M. Vin. Designing file systems for digital video and audio. In *Proc. Thirteenth ACM Symposium on Operating Systems Principles*, pages 81-94, Oct. 1991.
- [9] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the 13th Symposium on Operating Systems Principles*, pages 1-15. ACM, Oct. 1991.
- [10] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *Computer*, 27(3):17-28, Mar. 1994.
- [11] Seagate. *Product Manual - Hawk 1 Family SCSI-2 (Volume 1)*, Rev. D, 1994.
- [12] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS file system. In *Proc. 1996 USENIX Technical Conference*, pages 1-14. USENIX, Jan. 1996.
- [13] B. L. Worthington, G. R. Ganger, and Y. N. Patt. Scheduling for modern disk drives and non-random workloads. Technical Report CSE-TR-194-94, University of Michigan, Mar. 1994.
- [14] B. L. Worthington, G. R. Ganger, Y. N. Patt, and J. Wilkes. On-line extraction of SCSI disk drive parameters. In *Proc. ACM Sigmetrics Conference*, May 1995.

## Availability

The code used for these measurements and data obtained is available on the web at <http://www.isi.edu/netstation/zcav/> or on the author's home page at <http://www.isi.edu/~rdv/> or <http://alumni.caltech.edu/~rdv/>.

The author may be contacted via email at [rdv@isi.edu](mailto:rdv@isi.edu) or [rdv@alumni.caltech.edu](mailto:rdv@alumni.caltech.edu).

# A Revisitation of Kernel Synchronization Schemes

Christopher Small and Stephen Manley

*Harvard University*

{chris,manley}@eecs.harvard.edu

## Abstract

In an operating system kernel, critical sections of code must be protected from interruption. This is traditionally accomplished by masking the set of interrupts whose handlers interfere with the correct operation of the critical section. Because it can be expensive to communicate with an off-chip interrupt controller, more complex optimistic techniques for masking interrupts have been proposed.

In this paper we present measurements of the behavior of the NetBSD 1.2 kernel, and use the measurements to explore the space of kernel synchronization schemes. We show that (a) most critical sections are very short, (b) very few are ever interrupted, (c) using the traditional synchronization technique, the synchronization cost is often higher than the time spent in the body of the critical section, and (d) under heavy load NetBSD 1.2 can spend 9% to 12% of its time in synchronization primitives.

The simplest scheme we examined, disabling all interrupts while in a critical section or interrupt handler, can lead to loss of data under heavy load. A more complex optimistic scheme functions correctly under the heavy workloads we tested and has very low overhead (at most 0.3%). Based on our measurements, we present a new model that offers the simplicity of the traditional scheme with the performance of the optimistic schemes.

Given the relative CPU, memory, and device performance of today's hardware, the newer techniques we examined have a much lower synchronization cost than the traditional technique. Under heavy load, such as that incurred by a web server, a system using these newer techniques will have noticeably better performance.

## 1 Introduction

Although the BSD kernel is traditionally single-threaded, it needs to perform synchronization in the face of asynchronous interrupts from devices. In addition, while processing an interrupt from a device, the kernel needs to block delivery of new interrupts from that

device. This is traditionally accomplished by communicating with an off-chip interrupt controller and masking interrupts. Historically, the routines that perform this service are named *splintr*, where *intr* is the name of the interrupt to disable. A special routine, *splhigh*, is used to disable all interrupts.

Most, if not all, conventional hardware platforms allow device interrupts to be prioritized and assigned levels; a pending interrupt from a lower-level device can not interrupt the processing of an interrupt from a higher-level device. This method protects critical sections (by disabling interrupts from all devices) and device drivers (by disabling interrupts from the same device). Interrupts can then be prioritized by their frequency and the time required to handle them.

At Harvard we are in the process of developing a new operating system, the VINO kernel [Selt96]. In order to explore the space of kernel synchronization methods, we constructed analytic models of four basic interrupt handling schemes. We then used the NetBSD 1.2 kernel to analyze the performance of each scheme, by measuring:

- how often NetBSD enters critical sections,
- how long critical sections last,
- how often they are interrupted,
- the frequency of different types of interrupts, and
- the frequency with which these interrupts are delivered.

We found that critical sections are generally very short (usually less than four microseconds), that little total time is spent in critical sections, and that the synchronization overhead of three of the four basic schemes we examined is nominal. We also found that the overhead of the traditional scheme can be as large as 12% of CPU time.

On many system architectures interrupt processing is handled by an off-chip programmable interrupt controller. On a Pentium PC the time required to access the off-chip interrupt controller is quite high, approximately three hundred cycles (2.5 $\mu$ s on a 120MHz Pentium).

In addition to the off-chip interrupt controller, some CPUs include an on-chip instruction to mask all interrupts. On the Pentium this operation can be performed in five to ten cycles, two orders of magnitude

---

This research was supported by Sun Microsystems Laboratories.

more quickly than communicating with the interrupt controller.

If a critical section is short, the cost of going off-chip to disable and re-enable interrupts can be much greater than the time spent in the body of the critical section. For example, if the critical section adds an element to a linked list, it may only run for twenty or thirty cycles; if the kernel needs to go off-chip to synchronize, the time spent in the synchronization code is an order of magnitude more than the time spent in the body of the critical section. If we use the on-chip instructions to disable and re-enable interrupts, the synchronization cost is lowered to the point where it is less than that of the critical section code.

Optimistic synchronization techniques are particularly well-suited to short-duration critical sections that are infrequently interrupted. On entry to a critical section, instead of disabling interrupts, a global flag is set. If an interrupt is delivered while the flag is set, interrupts are only then disabled, and the interrupt is postponed until the flag is cleared. If no interrupt is delivered while the flag is set, the cost of synchronization is just the cost of setting and clearing the flag.

In this paper we model four schemes, based on two variables: where to mask interrupts (on-chip or off-chip) and how to mask them (pessimistically or optimistically).

We note that the techniques that use on-chip interrupt masking for synchronization may not be usable on multiprocessor systems. In symmetric multiprocessor systems, any processor may enter a critical section. Hence, any processor must be able to disable interrupts delivered to all processors. On this type of hardware the synchronization scheme must communicate either with the other processors (directly or through memory), or with an off-processor interrupt controller; in either case, it needs to go off-chip. An asymmetric multiprocessor (where only one processor runs operating system kernel code, and hence only one processor receives interrupts and enters critical sections) would be able to take advantage of the on-chip techniques we propose.

In the remainder of the paper we examine our four schemes in detail and compare their costs. We use our results to derive a fifth scheme which synthesizes the benefits of the others.

In Section 2, we discuss related work on synchronization. In Section 3, we propose our four schemes for handling interrupts. In Section 4 we discuss our experimental setup, and then in Section 5 measure the overhead of the four schemes. Section 6 examines the behavioral correctness of the on-chip schemes we propose, measuring the frequency of, and amount of time spent in, interrupt handlers. In Section 7 we discuss

our results and propose our new scheme, and conclude in Section 8.

## 2 Related Work

The interrupt handlers of real-time systems often run with interrupts disabled [Stan88], as in the V kernel [Berg86]. In this paper we explore similar techniques, using atomicity for synchronization, although we make no assumptions about real-time behavior of the system as a whole.

Synchronization does not require locking; other techniques have been proposed. For example, non-blocking synchronization [Herl90, Mass91, Green96], lock-free data structures [Wing92], and restartable atomic sequences [Bers92] are novel techniques proposed for synchronization. Some of these schemes require special hardware support, such as a *compare-and-swap* instruction, which is not available on all processors (e.g., the MIPS R3000 and the Intel i386). These techniques are designed to work in environments where it is expensive (or impossible) to disable interrupts, and work best in environments where contention is minimal. The cost and complexity of these schemes is higher than the on-chip schemes we propose, and our analysis shows that under typical loads, such complexity is unnecessary.

Similarly, Stodolsky et al. [Stod93] proposed decreasing the cost of synchronization by taking an *optimistic* approach. When the kernel makes a request to raise the global priority level, instead of communicating with the interrupt controller, the kernel saves the desired priority level in a variable. The critical section is then run, and on completion, the former priority level is restored. If no interrupts occurred during the critical section, the kernel was able to save the (high) cost of communicating with the interrupt controller, paying instead the (lower) cost of saving and restoring the logical priority level.

If an interrupt is delivered while the kernel is in a critical section, the interrupt handler compares the logical priority level with the interrupt priority level. If the interrupt is higher priority, it is handled immediately; if it is lower priority, the system queues the interrupt and falls back to the pessimistic scheme, setting the actual priority level to match the logical level. The cost of setting the global flag is quite low compared to the cost of communicating with the interrupt controller, so if the kernel is rarely interrupted while in a critical section the performance of the optimistic technique is superior to that of the standard pessimistic technique. In the case of a null RPC microbenchmark, they saw a 14% performance improvement.

Our work begins with the ideas developed by Stodolsky and adds a second dimension of comparison, the performance difference seen when masking interrupts on-chip and off-chip.

Mogul and Ramakrishnan [Mogul96] have explored a related issue. Interrupt processing can be split across multiple priority levels, a high-priority level that responds to the device interrupt and a low-priority level that processes the interrupt. When this is the case, it is possible for a flood of high priority device interrupts to cause the starvation of the low-priority processing. Mogul and Ramakrishnan refer to this state as *receiver livelock*. When the system reaches this state, their strategy is to disable interrupts and poll for new requests as the old ones are handled.

Their techniques are most appropriate in the face of *unsolicited I/O* from network and serial devices. The kernel has little or no control over how quickly data is sent to this type of device. Unlike a network device, a disk device only generates interrupts in response to a request from the kernel. If the kernel is receiving disk interrupts more quickly than it can process them, the kernel can reduce the load by queueing fewer disk requests. A natural feedback loop is present; if the kernel is bogged down handling old work, little new work will be started.

Mogul and Ramakrishnan's strategy is related to ours. They find that, at times, it is more efficient to ignore new interrupts in order to process the ones that have already arrived. However, they propose dynamically switching between two schemes based on the current system load; we propose choosing a single scheme for handling all interrupts. Their work is compatible with ours; none of the synchronization schemes we analyze preclude use of their techniques.

### 3 Synchronization Strategies

The first axis of our strategy space is a comparison of *optimistic* and *pessimistic* schemes, as studied by Stodolsky et al. The second axis of our space is a comparison of *off-chip* and *on-chip* interrupt management. This gives us four strategies to explore, as seen in Table 1.

In the following sections we describe each scheme in detail, sketch its synchronization functions, and discuss its costs and benefits.

|          | Pessimistic       | Optimistic       |
|----------|-------------------|------------------|
| Off-chip | <i>spl-pessim</i> | <i>spl-optim</i> |
| On-chip  | <i>cli-pessim</i> | <i>cli-optim</i> |

**Table 1: Synchronization Strategies Explored.** On the x86, NetBSD 1.2 uses *spl-optim*, the Linux 1.3.0 kernel uses *cli-pessim*, and BSD/OS 2.1 uses *spl-pessim*.

#### 3.1 Spl-Pessim

The *spl-pessim* scheme is named after the kernel “set priority level” function (*spl*), which was named after the PDP-11 instruction of the same name. When a critical section is entered, a subset of interrupts are disabled by communicating with the off-chip programmable interrupt controller (PIC).

```
crit_sec_enter()
    saved_mask = cur_mask
    PIC_mask(all)

crit_sec_leave()
    PIC_mask(saved_mask)

interrupt_handler(int intr_level)
    saved_mask = cur_mask
    PIC_mask(intr <= intr_level)
    handle interrupt
    PIC_mask(saved_mask)
```

The benefit of this scheme is fine-grained control of which interrupts are disabled. The cost is high per-critical section overhead.

#### 3.2 Spl-Optim

When a critical section is entered a variable is set to the logical priority level; the variable is restored on exit from the critical section. When an interrupt is delivered, the logical priority level is checked; if the interrupt has been masked, the interrupt is queued. The hardware interrupt mask is then set to be the logical mask (by communicating with the interrupt controller).

```
crit_sec_enter()
    in_crit_sec = true

crit_sec_leave()
    if (any queued interrupts)
        handle queued interrupts
    in_crit_sec = false

interrupt_handler(int intr_level)
    if (in_crit_sec
        || intr_level < cur_level)
        queue interrupt
    else
        saved_mask = cur_mask
        PIC_mask(intr <= intr_level)
        handle interrupt
        PIC_mask(saved_mask)
        handle queued interrupts
```

The benefit of this scheme is low per-critical section overhead, and fine-grained control over which interrupts are disabled.

The cost of this scheme is communication with the off-chip interrupt controller if an interrupt is delivered when the kernel is in a critical section, and a higher code complexity than the pessimistic schemes.

### 3.3 Cli-Pessim

The *cli-pessim* scheme is named after the x86 instruction that clears the interrupt flag, *cli*. When a critical section is entered, all interrupts are disabled. This scheme is structurally similar to the *spl-pessim* scheme, but instead of communicating with the PIC, on-chip instructions are used to disable and enable interrupts.

```
crit_sec_enter()
    disable_all_interrupts()

crit_sec_leave()
    enable_all_interrupts()

interrupt_handler(int intr_level)
    crit_sec_enter()
    handle interrupt
    crit_sec_leave()
```

The benefit of this scheme is a low per-critical section overhead. The cost is increased risk of dropped interrupts (if multiple interrupts are delivered during critical sections) and possible delay of high-priority interrupts while processing a lower priority interrupt or while the kernel is in a critical section.

### 3.4 Cli-Optim

When a critical section is entered, a global flag is set; it is cleared on exit from the critical section. When an interrupt is delivered, interrupts are disabled while the interrupt is processed.

```
crit_sec_enter()
    in_crit_sec = true

crit_sec_leave(int level)
    if (any queued interrupts)
        handle queued interrupts
    in_crit_sec = false

interrupt_handler(int intr_level)
    if (in_crit_sec)
        queue interrupt
    else
        disable_all_interrupts()
        handle interrupt
        enable_all_interrupts()
```

This scheme is very similar to the *spl-optim* scheme, but, as with the *cli-pessim* scheme, instead of communicating with the PIC, interrupts are disabled and enabled through the use of on-chip instructions.

The benefit of this scheme is low per-critical section overhead. Its cost is increased risk of lost interrupts if multiple interrupts are delivered during critical sections, and possible delay of high-priority interrupts while processing a lower priority interrupt or critical section. It also has a slightly higher code complexity than the *cli-pessim* scheme.

### 3.5 Comparison of Schemes

While a scheme's performance is important, we must also verify its correctness. Devices are designed to queue a small number of pending interrupts for a short period of time; if interrupts are disabled for a long period of time, it is possible that multiple interrupts will be merged together or lost, and data buffers can overflow with an accompanying loss of data. In some cases the loss of an interrupt is a performance issue, not a correctness issue (e.g., TCP/IP packets are retransmitted if dropped). However, we need to be sure that the system as a whole behaves correctly in the face of heavy load.

Fortunately, it is not necessary to test all four schemes for correctness. The *cli-pessim* scheme is the least responsive to external interrupts; if it behaves correctly in the face of heavy load, the other systems can do no worse.

Note that use of a *cli* scheme does not preclude assigning priorities to interrupts. Although all interrupts are disabled while the system is in a critical section, pending interrupts can still be assigned priority levels, and higher-priority interrupts will take precedence over lower-priority interrupts when interrupts are once again enabled. (In our implementation of the *cli-pessim* kernel we assign the same priorities to interrupts as are used by standard NetBSD1.2, the *spl-optim* kernel to which we compare it.)

The costs and benefits of the four strategies depend on a number of components. First, the cost of going off-chip to set the priority level needs to be measured, as does the cost of disabling interrupts on-chip. Second, the frequency with which interrupts are received, and the frequency with which an interrupt arrives while the kernel is in a critical section, must be measured. We must also determine the time spent in critical sections in order to learn whether a *cli-pessim* kernel will ignore interrupts for too long a period of time, with too high a risk of losing interrupts.

| Scheme            | Synchronization Overhead   |
|-------------------|--|
| <i>spl-pessim</i> | (number of critical sections entered) • (off-chip cost)  |
| <i>spl-optim</i>  | (number of critical sections entered) • (flag-setting cost) +<br>(number of critical sections interrupted) • (off-chip cost) |
| <i>cli-pessim</i> | (number of critical sections entered) • (on-chip cost)   |
| <i>cli-optim</i>  | (number of critical sections entered) • (flag-setting cost) +<br>(number of critical sections interrupted) • (on-chip cost)  |

**Table 2: Synchronization Scheme Overheads.** The model describing the overhead associated with the four schemes discussed. The costs are functions of five variables: critical sections entered, critical sections interrupted, off-chip cost, on-chip cost, and flag-setting cost.

The overhead of synchronization for each strategy can be described by a simple analytic model. The model must take into account:

- *number of critical sections entered*: number of times a critical section was entered, per second. This factor is important when computing the overhead for any of the schemes.
- *number of interrupted critical sections*: number of times a critical section was interrupted, per second. This is only a factor for the *optimistic* schemes.
- *off-chip cost*: the time required to communicate with the interrupt controller, first raising the priority level, then lowering it. This is only relevant for the *spl* schemes.
- *on-chip cost*: the time required to disable interrupts and re-enable them. This cost only applies to the *cli* schemes.
- *flag setting cost*: the time required to set the variable holding the mask. This cost is only a factor when using an *optimistic* scheme.

The equations that describe the overhead for each scheme are found in Table 2. In the following section we discuss our experimental setup, and then in Section 5 we measure the component costs and derive a total cost for each scheme.

## 4 Experimental Setup

In this section we discuss the kernels that we constructed and measured, and the hardware platform used.

### 4.1 Kernels

The x86 version of NetBSD 1.2, a derivative of 4.4BSD Lite, uses the *spl-optim* strategy for priority level management. We started with an off-the-shelf copy of NetBSD 1.2 for our *spl-optim* measurements. With this kernel we were able to measure the frequency with which critical sections are interrupted.

Starting with NetBSD 1.2 as a base, we developed a *cli-pessim* kernel. This kernel disables all interrupts any

time a critical section or interrupt handler is entered, and enables all interrupts when the critical section or interrupt handler finishes. The *cli-pessim* kernel allowed us to accurately measure the length of time spent in critical sections, the number of critical sections, and the time required to handle each type of interrupt.

## 4.2 Hardware

We ran our tests on an x86 PC with a 120MHz Pentium processor and PCI bus. The memory system consisted of the on-chip cache (8KB i + 8KB d), a 512KB pipeline burst off-chip cache and 64MB of 60ns EDO RAM. There was one IDE hard drive attached, a 1080 MB, 5400 RPM Western Digital WDC31000, with a 64KB buffer and an average seek time of 10ms. The system included a BusLogic BT946C PCI SCSI controller; attached to it was a 1033 MB, 5400 RPM Fujitsu M2694ES disk, with a 10ms average seek time and a transfer rate of 5MB/second, and a Sony 4x SCSI CD-ROM drive (model CDU-76S). The Ethernet adapter was a 10Mbps Western Digital Elite 16 with 16KB of 100ns RAM. The serial ports were buffered (16550 UART).

Pentium processors include an on-chip cycle counter, which enable very precise timing measurements. We read the counter at the start and at the end of each experiment; the difference was multiplied by the processor cycle time (8 1/3 nanoseconds) to obtain the elapsed time. The cost of reading the cycle counter is roughly 10 cycles; where significant, the timing cost has been subtracted from our measurements. We include code to read the cycle counter in the Appendix.

## 5 Synchronization Overhead

As shown in Table 2, the synchronization overhead of each scheme is a function of five variables: the off-chip priority setting cost, the on-chip priority setting cost, the flag-setting cost, the number of times a critical section is

entered, and the number of times an interrupt arrives while the system is in a critical section. In this section we measure each component to derive the synchronization overhead for the four schemes.

## 5.1 Critical Sections Per Second

We ran four tests to estimate the number of critical sections entered per second under heavy load. These tests supply the value used for the *number of critical sections entered* variable in the equations above. They measure the system under a mixed programming load, heavy web traffic, and high-speed serial traffic.

The first two tests are the result of running the Modified Andrew Benchmark [Ouster90] on a local file system (Andrew local) and an NFS-mounted file system (Andrew NFS). The Modified Andrew Benchmark consists of creating, copying, and compiling a hierarchy of files, and was designed to measure the scalability of the Andrew filesystem [How88].

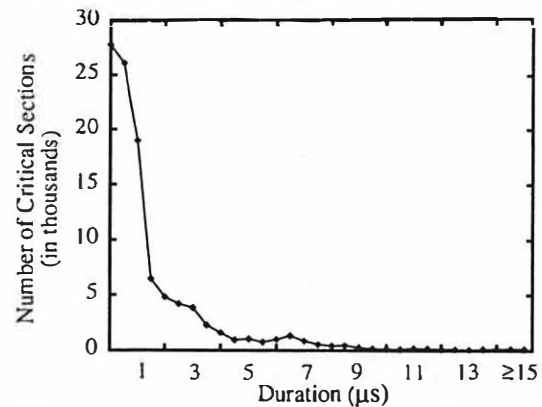
|                 | Mean<br>Critical<br>Sections<br>per sec<br>(std dev) | Max<br>Critical<br>Sections<br>per sec | Mean<br>Critical<br>Section<br>Duration |
|-----------------|--|--|---|
| Andrew<br>local | 36048<br>(60%)                                       | 72521                                  | 3.1 $\mu$ s                             |
| Andrew<br>NFS   | 25867<br>(68%)                                       | 72940                                  | 4.6 $\mu$ s                             |
| WebStone        | 34468<br>(31%)                                       | 82653                                  | 3.3 $\mu$ s                             |
| Serial 115.2    | 47762<br>(0.05%)                                     | 47807                                  | 0.6 $\mu$ s                             |

**Table 3: Critical Sections Per Second.** The mean and maximum number of critical sections entered, per second, measured with the *cli-pessim* kernel. We also measured the mean critical section duration, which is seen to be very short, especially relative to the cost of off-chip synchronization.

The third test was running the WebStone 2.0 benchmark from Silicon Graphics [SGI96]. WebStone was designed to test the performance of web servers under varying load. We installed the Apache 1.1.1 server on our test machine, and ran the WebStone clients (which generate http requests of the server) from a SparcStation 20. The maximum number of critical sections per second occurred with 50 client processes; we report the results from this test.

The fourth test measured the kernel's behavior while receiving a 64KB file via *tip(1)* at 115,200 bps. This test measures the behavior of the kernel when it is presented with a high rate of short-duration interrupts.

We instrumented the kernel to measure the duration of each critical section, using the Pentium cycle counter. A user-level process polled the kernel once per second to retrieve the number of critical sections and the time spent in critical sections since the last poll. We use these results to derive the numbers in Table 4.



**Figure 1: Critical Section Duration Distribution.** Distribution of critical section durations for Andrew NFS; the other distributions had the same shape.

We plotted the frequency of duration of critical sections for each test, and found that the distribution roughly follows the shape of an exponential distribution (see Figure 1 as an example; the other distributions had the same shape). The standard deviation of an exponential distribution is equal to the mean of the distribution, which in all measured cases is less than 5 $\mu$ s. If we use the mean as an estimate of the standard deviation, we find that most critical sections take less than 10 $\mu$ s; in the case of the Serial 115.2, we expect most to be less than 1.2 $\mu$ s.

## 5.2 Interrupted Critical Sections

In order to compute the overhead of the optimistic techniques we need to determine the percentage of critical sections that are interrupted. We reran the tests from Table 4 with the standard NetBSD 1.2 kernel (which uses the *spl-optim* scheme) and measured the number of critical sections interrupted per second. These results are shown in Table 4, on page 7.

For comparison with the results of Table 4, we include the Mean Critical Sections Per Second measured using this kernel; these results are very similar to the values seen in Table 4. As we expected (based on the short duration of critical sections), we saw that a very small number of critical sections are interrupted.

## 5.3 Synchronization Primitive Cost

The synchronization overhead parameters were measured by performing each operation pair (set and

|              | Mean Critical<br>Sections<br>per second<br>(std dev) | Interrupted<br>Critical<br>Sections<br>per second<br>(% intr) |
|--------------|--|---|
| Andrew local | 31904 (69%)  | 15 (0.05%)  |
| Andrew NFS   | 25172 (67%)  | 50 (0.2%)   |
| WebStone     | 33890 (33%)  | 300 (0.9%)  |
| Serial I15.2 | 45269 (8%)   | 140 (0.3%)  |

**Table 4: Interrupted Critical Sections.** We measured the number of critical sections entered per second on the *spl-optim* kernel, and the number of times per second critical sections were interrupted (which causes the optimistic kernel to revert to pessimistic synchronization behavior). We include the percentage of critical sections interrupted.

unset) in a tight loop 100,000 times and computing the mean and standard deviation. In all cases the standard deviation was less than 1% of the mean. The results of these measurements are given in Table 5.

Because these tests were run in a tight loop, a minimal number of cache misses and other side-effects are seen in these results, hence the synchronization times we use are somewhat optimistic. Cache misses could have a large percentage impact on the on-chip and flag-setting measurements, because these instruction sequences are very short. However, *because* they are so short, even if all of the instructions and data referenced by the code cause cache misses, the total absolute additional cost (in cycles) would be very small.

| Synchronization Primitive                        | Time                     |
|--|--------------------------|
| off-chip ( <i>spl-pessim</i> )                   | 2.5 $\mu$ s (304 cycles) |
| on-chip ( <i>cli-pessim</i> )                    | 0.18 $\mu$ s (21 cycles) |
| set flag ( <i>spl-optim</i> , <i>cli-optim</i> ) | 0.05 $\mu$ s (6 cycles)  |

**Table 5: Synchronization Primitive Cost:** Time required to set priority level off-chip, on-chip, and to set a global flag saving the logical priority level. The Pentium has a write buffer, so the flag-setting cost does not include time to write to main memory.

The synchronization cost of each of the four schemes is directly driven by the costs shown in Table 5. The *off-chip* cost is the synchronization cost of the *spl-pessim* kernel (304 cycles, 2.5 $\mu$ s), *on-chip* gives the synchronization cost the *cli-pessim* kernel (21 cycles, 0.18 $\mu$ s), and the flag setting cost is the synchronization cost of the optimistic kernels (6 cycles, 0.05 $\mu$ s). As we immediately see by comparing the costs in Table 5 with the critical section durations seen in Table 4, the mean critical section duration seen is less than twice the off-

chip synchronization cost, and more than three times the mean Serial I15.2 critical section cost.

## 5.4 Synchronization Overhead

Given the results in Table 3, Table 4, and Table 5, the synchronization cost and overhead for each of the four schemes can be computed. These costs are given in Table 6.

| Scheme            | Synchronization<br>Overhead |               |              |                 |
|-------------------|-----------------------------|---------------|--------------|-----------------|
|                   | Andrew<br>local             | Andrew<br>NFS | Web<br>Stone | Serial<br>I15.2 |
| <i>spl-pessim</i> | 9.0%                        | 6.5%          | 8.6%         | 11.9%           |
| <i>spl-optim</i>  | 0.2%                        | 0.1%          | 0.3%         | 0.3%            |
| <i>cli-pessim</i> | 0.7%                        | 0.5%          | 0.6%         | 0.9%            |
| <i>cli-optim</i>  | 0.2%                        | 0.1%          | 0.2%         | 0.2%            |

**Table 6: Synchronization Scheme Overhead.** Percentage of time spent synchronizing critical sections using the described techniques. Computed using equations in Table 2 and measurements from Table 3, Table 4, and Table 5.

We see that with the large number of critical sections per second seen under the Serial I15.2 test, the *spl-pessim* kernel can spend nearly 12% of its time in synchronization primitives. Even under the lightest measured load, the system spends more than 6% of its time in synchronization primitives.

Another thing to note is that the other three techniques all have an overhead that is an order of magnitude less than that of the *spl-pessim* technique. Even though the *spl-optim* technique goes off-chip for synchronization, because so few critical sections are interrupted (from 0.05% to 0.9%) the difference is negligible (between 0.2%-0.7%).

## 6 Correctness

With the knowledge that the absolute overhead of the three proposed schemes is very low, we must be sure that none of the schemes allow interrupts to be lost or overly delayed.

The optimistic schemes do not delay high-priority interrupts, nor does the *spl-pessim* scheme; the *cli-pessim* scheme is the only one that masks all interrupts while in a critical section or interrupt handler. If the *cli-pessim* scheme can operate correctly, the other schemes will also operate correctly.

In Table 3 we saw that the mean critical section duration is between 0.6 and 4.6 $\mu$ s. We are concerned that handling a device interrupt could take much longer, especially with programmed I/O devices, such as the

| Test                     | Duration of Test (seconds) | Total Interrupts | Interrupts per second | Mean time to handle interrupt (std dev) | Min time to handle interrupt | Median time to handle interrupt | Max time to handle interrupt |
|--------------------------|----------------------------|------------------|-----------------------|---|------------------------------|---------------------------------|------------------------------|
| <i>ide disk</i>          | 3.2s                       | 1027             | 321                   | 1519 $\mu$ s (7%)                       | 15 $\mu$ s                   | 1545 $\mu$ s                    | 2003 $\mu$ s                 |
| <i>scsi disk</i>         | 8.8s                       | 553              | 61                    | 65 $\mu$ s (15%)                        | 16 $\mu$ s                   | 64 $\mu$ s                      | 86 $\mu$ s                   |
| <i>scsi cd-rom</i>       | 13.4s                      | 3931             | 293                   | 27 $\mu$ s (7%)                         | 20 $\mu$ s                   | 26 $\mu$ s                      | 40 $\mu$ s                   |
| <i>floppy</i>            | 64.2s                      | 506              | 8                     | 169 $\mu$ s (57%)                       | 4 $\mu$ s                    | 200 $\mu$ s                     | 271 $\mu$ s                  |
| <i>serial-300 (8K)</i>   | 274s                       | 8221             | 30                    | 9 $\mu$ s (5%)                          | 9 $\mu$ s                    | 9 $\mu$ s                       | 12 $\mu$ s                   |
| <i>serial-38.4 (64K)</i> | 17.4s                      | 8195             | 471                   | 27 $\mu$ s (2%)                         | 19 $\mu$ s                   | 27 $\mu$ s                      | 42 $\mu$ s                   |
| <i>serial-115 (64K)</i>  | 5.8s                       | 8176             | 1410                  | 27 $\mu$ s (4%)                         | 11 $\mu$ s                   | 27 $\mu$ s                      | 54 $\mu$ s                   |
| <i>ethernet-small</i>    | 2.4s                       | 1003             | 418                   | 78 $\mu$ s (25%)                        | 44 $\mu$ s                   | 78 $\mu$ s                      | 370 $\mu$ s                  |
| <i>ethernet-large</i>    | 3.4s                       | 1703             | 501                   | 487 $\mu$ s (8%)                        | 45 $\mu$ s                   | 491 $\mu$ s                     | 1194 $\mu$ s                 |

Table 7: Behavior of *spl-optim* kernel during device tests. Frequency of device interrupts and time to handle an interrupt from each device was measured. Although the spread between minimum and maximum is large, the median is in all cases other than *floppy* is within 2% of the mean. Each test measured the reception of several thousand interrupts.

IDE disk, which copy data from the device to the kernel one word at a time.

We are concerned that high-frequency interrupts are not overly delayed while in long-duration interrupt handlers. We gathered data on both long-duration handlers (IDE disk interrupts) and high-frequency interrupts (serial lines at 115.2Kbps).

We measured the time required to process interrupts from a variety of devices attached to our test system. Our tests involve reading from or writing to each device and measuring the amount of time required to perform the test and the number of interrupts delivered while performing the test. We ran the following tests:

- *ide disk*: write 8MB to IDE disk.
- *scsi disk*: write 8MB to SCSI disk.
- *scsi cd-rom*: read 8MB from SCSI CD-ROM.
- *serial-300*: read 8KB (via tip) at 300bps.
- *serial-38.4*: read 64KB (via tip) at 38.4Kbps.
- *serial-115.2*: read 64KB (via tip) at 115.2Kbps.
- *floppy*: read 1MB from raw floppy disk.
- *ether-small*: flood ping of 1000 64 byte packets.
- *ether-large*: flood ping of 1000 1024 byte packets.

The results of these tests are shown in Table 7. The mean time to handle an interrupt from a particular device is useful, although we found that there can be considerable variation. For each test we also include the minimum, maximum, and median interrupt handling time.

## 6.1 IDE, SCSI, and Floppy Interrupts

As mentioned above, the *ide disk* test performs programmed I/O (PIO), i.e., data is copied by the CPU one byte or word at a time between the disk controller and main memory. This is very CPU intensive, and can be quite slow. By comparison, the SCSI disk controller uses direct memory access (DMA) to copy data directly from the controller to main memory without the intervention of the CPU. Each *ide disk* interrupt took substantially longer than a *scsi disk* interrupt (1519 $\mu$ s vs. 65 $\mu$ s).

Surprisingly, although the specifications of the IDE and SCSI disks is very similar, it took nearly three times as long to transfer 8MB to the SCSI disk as it did to transfer it to the IDE disk. We believe that this is an artifact of the NetBSD 1.2 disk drivers; we saw similar performance with the generic NetBSD 1.2 kernel and our *cli-pessim* kernel. We do not believe that this reflects the underlying performance of the disk or the controller; under BSD/OS 2.1 the performance of the two disks was much closer (in fact, the SCSI disk was about 10% faster than the IDE disk).

The mean time required to process an IDE interrupt (1.5ms) is substantially greater than any other interrupt processing time we measured. We discuss the possibility of timing conflicts between IDE interrupts and other interrupts in Section 6.4.

The standard deviation of *floppy* interrupt processing time is quite large (57%). It is caused by the several different types of interrupts generated by the floppy device, from seek completion notification (4 $\mu$ s),

to transfer completion notification (25 $\mu$ s), to data transfer (250 $\mu$ s). The maximum time seen, 271 $\mu$ s, is short enough that we are not concerned about floppy interrupts.

## 6.2 Serial Interrupts

We see that a *serial-300* interrupt takes one third as long as a *serial-38.4* interrupt; we attribute this to the fact that multiple characters are transferred on each *serial-38.4* interrupt, as evidenced by the fact that 1/8 as many interrupts were generated per kilobyte by the latter. (Please note that the *serial-300* test transferred 8KB, where the other two tests transferred 64KB.)

When we tested higher-rate serial connection (at 115.2 bps) we saw roughly the same number of interrupts as we saw in the *serial-38.4* test (8195 vs. 8176) and the same mean interrupt handling time (27 $\mu$ s), implying the same number of characters processed per interrupt, but the total test took one third as long. The high rate of interrupts during the *serial-115.2* test may conflict with the large amount of time required to process an *ide-disk* interrupt; we discuss this possibility in Section 6.4.

## 6.3 Network Interrupts

As stated above, our network tests were taken using a 10Mbps Ethernet board. We ran three tests: flood (continuous) ping, UDP, and TCP. We saw no significant variation in interrupt processing time as a function of protocol (ping/IP, UDP, and TCP), and hence include only the ping/IP results.

Interrupt processing time is independent of protocol because the higher-level protocol processing (where we would see differences between TCP, UDP, and ping) does not take place when the packet arrives; instead, the packet is queued for processing by a higher-level “soft” interrupt handler which runs once the hardware interrupt has completed.

We ran additional tests with varying packet sizes. We saw the expected relationship between packet size and processing time (which more or less scaled linearly with packet size).

Faster network technologies (e.g., 100Mbps Ethernet and ATM) would generate interrupts much more frequently than our 10Mbps Ethernet interface. However, as we see below, the rate at which the serial device generates interrupts is sufficient to preclude use of the *cli* schemes. Based on this, the interrupt generation rate of ATM is a moot point.

## 6.4 Summary

When examining the data gathered for each interrupt, we found that although there was often a significant difference between the minimum and maximum, in most cases the standard deviation was relatively small, and the median was close to the mean.

Our results show that in our environment it takes little time to handle an interrupt from any of our devices. No interrupt seen took more than 2ms to service, and in most cases the interrupt processing time was several orders of magnitude less than that.

However, we see from Table 7 that interrupts can occur very frequently. For example, the *serial-115.2* interrupts arrived at 1400Hz (every 700 $\mu$ s). We must examine the consequences of delaying the processing of these interrupts.

As was discussed in Section 2, we can partition interrupts into two classes: *solicited* and *unsolicited*. *Solicited interrupts* come in response to requests from the CPU (e.g., in response to a disk request). *Unsolicited interrupts* are generated externally, and are not under the control of the system (e.g., serial and network traffic). The rate of solicited interrupts are under the control of the system; when a solicited interrupt is received, the system spends its time processing the interrupt rather than generating more requests. The system can thus control the rate at which solicited interrupts are generated.

The system does not control the rate at which unsolicited interrupts are generated. However, the system can indirectly impact the rate at which they are generated, either in software (by not sending acknowledgments to network packets that are dropped or processed in time), or in hardware (by use of serial-line flow control).

To block serial interrupts it is necessary for the system to communicate with the device, and to be aware that it is necessary to slow the interrupt rate. If serial interrupts are overly delayed, the system will not be aware of the possibility of overflow, and hence will not be able to stem the tide in time.

From our measurements, the longest that an interrupt will be delayed is 2ms (the longest time spent handling an *ide-disk* interrupt). The *serial-115.2* device receives characters every 70 $\mu$ s (115.2 bits per sec = 14,400 bytes per sec = one byte every 70 $\mu$ s). This means that up to 28 characters (2ms / 70 $\mu$ s = 28) could arrive while an Ethernet interrupt is being processed. Although the serial port is buffered, the buffer only holds 16 characters. At this rate, characters could easily be lost on the serial line while processing an IDE interrupt.

If we could eliminate programmed I/O devices, the *cli* schemes would work. However, newer devices with

a high interrupt rate (e.g., ATM network controllers), combined with the possibility of slow interrupt handlers shift the balance against the *cli* techniques. Although, in some hardware combinations, the *cli* schemes would work, many common configurations could easily lead to loss of data.

The failure of the strict *cli* schemes leads us to propose a new, hybrid scheme, *cli-spl-pessim*, which is described in Section 7. The *cli-spl-pessim* scheme combines the low cost of the *cli* schemes for critical section synchronization with the interrupt prioritization of the *spl* schemes.

## 7 Analysis and Proposal

Our results show that under the benchmark workloads the absolute cost of synchronization using the optimistic techniques is low, less than 0.4%. The traditional *spl-pessim* scheme has a much higher cost, over 6% in all tested cases.

The performance improvement we see is less than that seen by Stodolsky et al. with their *spl-optim* scheme (14%). We attribute this in part to the differences between our benchmarks and theirs; their test consisted of a highly optimized null RPC, which has a single critical section. Because the null RPC took very little time (2140 cycles), an improvement in critical section synchronization had a large performance impact. Under the more varied loads that we measured, the kernel spends a much lower percentage of its time in synchronization code.

For the *cli* schemes to work, critical sections and interrupt handlers must complete their work quickly. These schemes completely disable interrupts during a critical section, so a long-running interrupt handler can delay the delivery of interrupts for a long period of time, increasing the likelihood of data loss. The combination of devices whose interrupt handlers require a long time to service (e.g., IDE disks with programmed I/O) with low latency requirements (e.g., very fast serial ports) is a worst-case scenario for the *cli* scheme.

### 7.1 Proposal

We find the simplicity of the *pessim* schemes and the low cost of the *cli* schemes appealing. This leads us to propose a fifth scheme, *cli-spl-pessim*, which combines the *cli* technique for critical sections with the prioritization of the *spl* technique for interrupt handlers. In this scheme, the *cli* and *sti* instructions are used to mask interrupts while the kernel is in a critical section; because critical sections are quite short, this will not overly delay interrupt delivery. When handling an interrupt, the kernel communicates with the off-chip

interrupt controller, disabling the delivery of equal and lower-priority interrupts. Because

- interrupts are delivered much less frequently than critical sections are entered,
- interrupt handlers take much longer than critical sections, and
- on our hardware platform, when an interrupt is delivered it is necessary to communicate with the off-chip controller,

the additional performance impact of going off-chip will be minimal and more than made up for by the increase in system robustness. In the format of Section 3, we include the following pseudo-code for *cli-spl-pessim*:

```
void crit_sec_enter(int level)
    disable all interrupts

void crit_sec_leave(int level)
    enable all interrupts

void interrupt_handler
    prev_level = cur_interrupt_level
    PIC_mask(interrupt_level)
    handle_interrupt
    PIC_mask(prev_level)
```

The benefit of this scheme is the low cost of synchronization in critical sections. In addition, because we use an *spl* scheme for synchronizing handlers, a low-priority handler will be interrupted by the arrival of a high-priority interrupt.

Because our model includes only the cost of synchronization of critical sections, and disregards the cost of the synchronization overhead in interrupt handlers, using our model the cost of *cli-spl-pessim* is identical to that of the *cli-pessim* scheme, with (as shown in Table 6) sub-1% synchronization overhead for the system loads that we studied.

With the performance characteristics and simplicity of the *cli-pessim* scheme, without its incorrect behavior under heavy load, we plan to use the *cli-spl-pessim* scheme in the implementation of our new operating system kernel.

### 7.2 Related Schemes

Although we have described NetBSD as using the *spl-optim* technique, it also supports a *cli-optim*-style technique for short duration interrupt handlers.

The Linux<sup>1</sup> operating system uses a scheme similar to *cli-spl-pessim*. The *cli* and *sti* instructions are used to synchronize critical sections (with the concomitant high performance), and, like NetBSD 1.2, short-duration

1. We examined the source code of version 1.3.0.

interrupt handlers are run with all interrupts disabled. Long-duration handlers are either run with no protection from interrupts (which increases their complexity) or wake a kernel task to handle the processing of the interrupt, and then return. Although this scheme efficiently synchronizes critical sections, we believe that the increased latency and complexity of waking a separate task argue against use of this technique.

## 8 Conclusions

In this paper we have shown that it is worthwhile to rethink how interrupts and synchronization are handled on modern hardware<sup>2</sup>.

Our results have driven the design of our new kernel, where we use the proposed *cli-spl-pessim* scheme. This scheme provides us with the simplicity of the pessimistic schemes we describe, with the low overhead of the optimistic schemes.

Comparing our scheme to those used in earlier systems, we are reminded of how quickly hardware changes: the CPU and I/O bus of today is substantially faster than the one available in 1993 [Stod93], and much, much faster than the one available on the VAX or the PDP-11 [Leff89]. Nevertheless, developers of newer systems [Cust93] find themselves re-using old, complex techniques to solve a problem that may no longer exist.

## Status and Availability

All code discussed in this paper (benchmark tests and patch files for a *cli-pessim* NetBSD 1.2) is available at <http://www.eecs.harvard.edu/~chris>.

## Acknowledgments

Our thanks to our advisor, Margo Seltzer, for her invaluable feedback, wordsmithing, and grasp of the subtleties of “that” and “which.” We thank our reviewers, especially Mike Karels, our shepherd, for providing very helpful feedback. And thanks to Brad Chen, whose previous work on fast interrupt priority management helped inspire this work.

## Bibliography

- [Berg86] Berglund, E., “An Introduction to the V-System”, *IEEE Micro* 10, 8, 35–52 (August 1986).
- [Bers92] Bershad, B., Redell, D., Ellis, J., “Fast Mutual Exclusion for Uniprocessors,” *ASPLOS V*, 223–233, Boston, MA (1992)
- [Cust93] Custer, H., *Inside Windows NT*, Microsoft Press, Redmond, WA, 218–221 (1993).
- [Green96] Greenwald, M., Cheriton, D., “The Synergy Between Non-blocking Synchronization and Operating System Structure,” *Second Symposium on Operating Systems Design and Implementation*, 123–136, Seattle, WA (1996).
- [Herl90] Herlihy, M. “A Methodology for Implementing Highly Concurrent Data Structures,” *Second ACM Symposium on Principles and Practice of Parallel Programming*, 197–206 (1990).
- [How88] Howard, J., Kazar, M., Menees, S., Nichols, D., Satyanarayanan, M., Sidebotham, R., West, M., “Scale and Performance in a Distributed File System,” *ACM Transactions on Computer Systems* 6, 1 51–81 (1988).
- [Intel95] *Pentium Processor Family Developer's Manual, Volume 3: Architecture and Programming Manual*, Intel Corporation, Mt. Prospect, IL (1995).
- [Leff89] Leffler, S., McKusick, M. K., Karels, M., Quarterman, J., *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley Publishing, Reading, MA (1989).
- [Mass91] Massalin, H., Pu, C., “A Lock-Free Multiprocessor OS Kernel,” Technical Report CUCS-005-91, Department of Computer Science, Columbia University (1991)
- [Mogul96] Mogul, J., Ramakrishnan, K. K., “Eliminating Receiver Livelock in an Interrupt-Driven Kernel,” *USENIX Technical Conference*, 99–111, San Diego, CA (1996).
- [Ouster90] Ousterhout, John, “Why Aren't Operating Systems Getting Faster as Fast as Hardware,” *USENIX Summer Conference*, 247–256, Anaheim, CA (1990)
- [Selt96] Seltzer, M., Endo, Y., Small, C., Smith, K., “Dealing With Disaster: Surviving Misbehaving Kernel Extensions,” *Proceedings of the Second OSDI*, Seattle, WA (1996).
- [SGI96] Silicon Graphics Inc., “Webstone: World-Wide Web Server Benchmarking,” <http://www.sgi.com/Products/WebFORCE>.
- [Stan88] Stankovic, J., Ramamritham, K., “A New Hard Real-Time Kernel”, *Hard Real-Time Systems*, IEEE, 361–370 (1988).
- [Stod93] Stodolsky, D., Chen, J. B., Bershad, B., “Fast Interrupt Priority Management in Operating System Kernels,” *USENIX Symposium on Microkernels and Other Kernel Architectures*, 105–110, San Diego, CA (1993).
- [Wing92] Wing, J., Gong, C., “A Library of Concurrent Objects,” CMU CS TR 90-151, School of Computer Science, Carnegie Mellon University (1992).

---

2. At least, modern by comparison with the PDP-11.



# Porting UNIX\* to Windows NT

David G. Korn (dgk@research.att.com)

*AT&T Laboratories  
Murray Hill, N. J. 07974*

## Abstract

The Software Engineering Research department at Murray Hill writes and distributes several widely used development tools and reusable libraries that are portable across virtually all UNIX platforms.<sup>[1]</sup> To enhance reuse of these tools and libraries, we want to make them available on systems running Windows NT<sup>[2]</sup> and/or Windows 95<sup>[3]</sup>. We did not want to support multiple versions of these libraries, and we wanted to minimize the amount of conditionally compiled code.

This paper describes an effort of trying to build a UNIX interface layer on top of the Windows NT and Windows 95 operating system. The goal was to build an open environment rich enough to be both a good development environment and a suitable execution environment. This meant that the overhead needed to be small enough so that there was no incentive to program to the native operating system directly. The openness meant that the complete facilities of the native operating system were accessible through this environment.

The result of this effort is a set of libraries, headers, and utilities that we collectively refer to as UWIN. UWIN contains nearly all the X/Open Release 4<sup>[4]</sup> headers, interfaces and commands. We discuss alternative porting strategies, commercial products, design goals, problems that had to be overcome, and the current status. Some performance measurements of the current system are presented here.

## 1. INTRODUCTION

The marketplace has dictated the need for software applications to work on a variety of operating system platforms. Yet, maintaining separate source code versions and development environments creates

additional expense and requires more programmer training.

One way to lower this cost is to use a middleware layer that hides the differences among the operating systems. The problem with this approach is that it forces you to program to a non-standard, and often proprietary, interface. In addition, it often limits you to the least common denominator of features of the different operating systems.

An alternative is to build a middleware layer based on existing standards. This has been the approach followed by IBM with the introduction of OpenEdition<sup>[5]</sup> for the MVS operating system, URL <http://www.s390.ibm.com/products/oe>. OpenEdition is X/Open compliant so that a large collection of existing software can be transported at little cost.

Windows NT is an operating system developed by Microsoft to fill the needs of the high-end market. It is a layered architecture, designed from the ground up, built around a microkernel that is similar to Mach.<sup>[6]</sup> One or more *subsystems* can reside on top of the microkernel which gives Windows NT the ability to run different logical operating systems simultaneously. For example, the OS/2 subsystem allows OS/2 applications to run on Windows NT. The most important subsystem that runs on Windows NT is the WIN32 subsystem. The WIN32 subsystem runs all applications that are written to the WIN32 Application Programming Interface (API)<sup>[7]</sup>. The API for the WIN32 subsystem is also provided with Windows 95, although not all of the functions are implemented. In most instances binaries compiled for Windows NT that use the WIN32 API will also run on Windows 95.

The POSIX subsystem allows applications that are strictly conforming to the IEEE POSIX 1003.1

\* UNIX is a registered trademark, licensed exclusively through X/Open, Limited.

operating system standard<sup>[8]</sup> to run on Windows NT. Since the POSIX standard contains most of the standard UNIX system call interface, many UNIX utilities are simple to port to any POSIX system. Because most of our tools require only the POSIX interface, we thought that it would be sufficient to port them to the POSIX subsystem of Windows NT. We were wrong for the reasons described in the next section.

We investigated alternative strategies that would allow us to run programs on both UNIX and Windows NT based systems. After looking at all the alternatives, we decided to write our own library that would make porting to Windows NT and Windows 95 easy. We spent three months putting together the basic framework and getting some tools working. Realizing that the task was larger than a one person project, we contracted a small development team of 2 or 3 to do portions of the library, packaging, and documentation. This paper will discuss porting alternatives, the goals for our library, the issues that need to be addressed, and the implementation of our POSIX library. Finally, we present some performance results and future directions.

## 2. ALTERNATIVE STRATEGIES

Six basic strategies can be employed to port existing UNIX based applications to Windows NT. The first strategy is to rewrite the code using the WIN32 API. This strategy makes sense if there are no requirements to continue to run on a UNIX system. Otherwise, this strategy will either require two sets of source (which will most likely be too expensive to maintain) or the use of a WIN32 emulation library that runs on UNIX platforms. There are at least two vendors that have WIN32 API libraries for UNIX systems. We ruled out this approach because of the effort to rewrite the code to the WIN32 API and because the WIN32 API is more complex than the X/Open API.

The second strategy is to use the Microsoft C library. Microsoft supplies a library of routines that are similar to their UNIX counterparts. You could then make modifications to your application as necessary where the calls differ from the UNIX call. This strategy has been used by at least one commercial UNIX tools vendor to port GNU based tools to Windows NT. While this strategy is appropriate for some applications, other applications may require much work to overcome some subtle differences. In addition, the resulting code may have a large amount of conditionally compiled code that is hard to test

and maintain.

A third strategy would be to rewrite the code using a *framework* which provides a virtual system interface. There are several vendors that offer object-oriented application layer interfaces that encapsulate the operating system and therefore enable applications to work on multiple systems. There are three drawbacks to this approach. First of all, it requires a large up front investment. Secondly, you will be locked into the vendors' libraries and not able to take advantage of savings that result from competition. Finally, you will likely be restricted to the intersection of features available on the underlying platforms.

A fourth strategy is to port the application to the POSIX subsystem of Windows NT. The POSIX subsystem can run any strictly conforming IEEE POSIX application program. This strategy should not require major investment, and any investment that you make should increase the portability of your application to other POSIX conforming systems. Unfortunately, this is not a viable alternative for most applications. Microsoft has made the POSIX subsystem as useless as possible by making it a closed system. There is no way to access functionality outside of the 1990 POSIX 1003.1 standard from within the POSIX subsystem, either at the library level or at the command level. Thus, you cannot even invoke the Microsoft C compiler from within the POSIX subsystem. However, since you can invoke POSIX commands from the WIN32 subsystem, it is possible to port some stand alone programs to the POSIX subsystem. For example, we ported the `pax` utility, the POSIX 1003.2<sup>[9]</sup> replacement for `cpio` and `tar`, to Windows NT, and it can be invoked from any WIN32 program. Softway System, Inc., URL <http://www.softway.com>, has an agreement with Microsoft to enhance the POSIX subsystem so that they can achieve POSIX 1003.2 conformance. Softway claims that they will open up the POSIX subsystem so that it can access WIN32 applications. Even if the POSIX subsystem on Windows NT is opened up, the POSIX subsystem is not available for Windows 95.

The fifth strategy is to use an existing POSIX or X/Open library that runs in the WIN32 subsystem. At the time that we began this effort, we were aware of two vendors that sell such libraries but as discussed later, these products were less than satisfactory. In addition, Steve Chamberlain at Cygnus has started writing a POSIX interface for

Windows NT and Windows 95, but it appears as if his goals are less ambitious than ours. URL <http://www.cygnum.com/misc/gnu-win32/>.

A sixth and final strategy would be to write your own POSIX library using the WIN32 API. After investigating the other alternatives, this is what we decided to do. We are convinced that this was the best strategy for us, since we believe that it resulted in a better implementation than the two commercial products described later, and because it eliminates the need to pay licensing fees for each copy of each product that uses the library. The availability of source code makes it possible to provide adequate support.

### 3. GOALS

We wanted our software to work with Windows 3.1, Windows 95, and Windows NT. A summer student wrote a POSIX library for Windows 3.1 and we were able to port a number of our tools. However, the limited capabilities of Windows 3.1 made it a less than desirable platform. We instead focused our goals on Windows NT and Windows 95. We decided to use only the WIN32 API for our library so that the library would work on Windows 95 and so that all WIN32 interfaces would be available to applications.

Initially, our goal was to provide the IEEE POSIX.1 interface with a library. This would be sufficient to run `ksh` and about eighty utilities that we had written. It soon became obvious that this wasn't enough for many applications. Most real programs use facilities that are not part of this standard such as sockets or IPC.

We needed to provide a character based terminal interface so that curses based applications such as `vi` could run. After the initial set of utilities was running, we wanted to get several socket based tools working. Several projects at AT&T that became interested in using our libraries, required the System V IPC facilities. The S graphics system<sup>[10]</sup> and `ksh-93`<sup>[11]</sup> required runtime dynamic linking. As the project progressed, the need for privileged users, such as `root` on UNIX systems, surfaced. We decided that it was important to have `setuid` and `setgid` capabilities. It soon became clear that we needed full UNIX functionality and we set our goal on X/Open Release 4 conformance.

We needed to have a complete set of UNIX development tools since we didn't want to get into the business of rewriting makefiles or changing build

scripts. Most code written at AT&T, including our own, uses `nmake`<sup>[12]</sup>, (no relation to the Microsoft `nmake`), but we also wanted to be able to support other make variants. We didn't want to do manual configuration on tools that have automatic configuration scripts.

One important goal that we had from the beginning was to not require WIN32 specific changes to the source to get it to compile and execute. The reason for this is that we wanted to be able to compile and execute UNIX programs without having to understand their semantics. In addition we wanted to limit the number of new interfaces functions and environments variables that we had to add to use our library. It is difficult to manage more than one or two environment variables when installing a new package.

Another goal that we had was to provide a robust set of utilities with minimal overhead. If utilities written to the X/Open API were noticeably slower than the same utilities written to the native WIN32 API, then they were likely to be rewritten making our library unnecessary in the long run.

A final and important goal was interoperability with the native Windows NT system. Integration with the native system not only meant that we could use headers and libraries from the native system, but that we could pass environment variables and open file descriptors to commands written with the native system. There couldn't be two unrelated sets of user ids and separate passwords. If write permission were disabled from the UNIX system, then there should be no way to write the file using facilities in the native system and vice versa.

We have not as yet achieved all of our goals, but we think that we are close. We are in the process of running the X/Open conformance tests to verify compliance with the X/Open API's. The rest of the paper will discuss some of the issues we needed to deal with and our solutions.

### 4. PROBLEMS TO SOLVE

The following problems need to be understood and dealt with in porting applications to Windows NT. These are some of the issues that need to be addressed by POSIX library implementations. Section 6 describes how UWIN solved most of these problems.

## 4.1 Windows NT File Systems

Windows NT supports three different file systems, called FAT, HPFS, and NTFS. FAT, which stands for File Access Table, is the Windows 95 file system. It is similar to the DOS file system except that it allows long file names. There is no distinction between upper and lower case although the case is preserved. HPFS, which stands for High Performance File System, was designed for OS/2. NTFS, the native NT File System, is similar to the Berkeley file system.<sup>[13]</sup> It allows long file names (up to 255 characters) and supports both upper and lower case characters. It stores file names as 16 bit Unicode names.

The file system namespace in Win32 is hierarchical as it is in UNIX and DOS. A pathname can be separated by either a / or a \. Like DOS, and unlike UNIX, disk drives are specified as a colon terminated prefix to the path name, so that the pathname `c:\home\dgk` names the file in directory `\home\dgk` on drive `c:`. Many UNIX utilities expect only / separated names, and expect a leading / for absolute pathnames. They also expect multiple /'s to be treated as a single separator.

Even though NTFS supports case sensitivity for file names, the WIN32 API has no support for case sensitivity for directories and minimal support for case sensitivity for files, limited to a `FILE_FLAG_POSIX_SEMANTICS` creation flag for the `CreateFile()` function. Certain characters such as `*`, `?`, `>`, `|`, `:`, `"`, and `\`, cannot be used in filenames created or accessed with the WIN32 API. The names, `aux`, `com1`, `com2`, `nul`, and filenames consisting of these names followed by any suffix, cannot be created or accessed in any directory through the WIN32 API.

Because Windows 95 doesn't support execute permission on files, it uses the `.exe` suffix to decide whether a file is an executable. Windows NT doesn't require this suffix, but some NT utilities, such as the DOS command interpreter, require the `.exe` suffix.

## 4.2 Line Delimiters

Windows NT uses the DOS convention of a two character sequence `<cr><n1>` to signify the end of each line in a text file. UNIX uses a single `<n1>` to signify end of line. The result is that file processing is more complex than it is with UNIX. There are separate modes for opening a file as text and binary with the Microsoft C library. Binary mode treats the file as a sequence of bytes. Text mode strips off

each `<cr>` in front of each new-line as the file is read, and inserts a `<cr>` in front of each `<n1>` as the file is written. Because the number of characters read doesn't indicate the physical position of the underlying file, programs that keep track of characters read and use `lseek()` are likely to not work in text mode. Fortunately, many programs that run on Windows NT do not require the `<cr>` in front of each `<n1>` in order to work. This difference turned out to be less of a problem that we had originally expected.

## 4.3 Handles vs. file descriptors

The WIN32 API uses *handles* for almost all objects such as files, pipes, sockets, processes, and events, and most handles can be *duped* within a process or across process boundaries. Handles can be inherited from parent processes. Handles are analogous to file descriptors except that they are unordered, so that a per process table is needed to maintain the ordering.

Many handles, such as pipe, process, and event handles, have a `synchronize` attribute, and a process can wait for a change of state on any or all of an array of handles. Unfortunately, socket handles do not have this attribute. One of the few novel features of WIN32 is the ability to create a handle for a directory with the `synchronize` attribute. This handle changes state when any files under that directory change. This is how multiple views of a directory can be updated correctly in the presence of change.

## 4.4 Inconsistent Interfaces

The WIN32 API handle interface is often inconsistent. Failures from functions that return handles return either 0 or -1 depending on the function. The `CloseHandle()` function does not work with directory handles. The WIN32 API is also inconsistent with respect to calls that take pathname arguments and calls that take handles. Some functions require the pathname and others require the handle. In some instances, both calls exist, but they behave a little differently.

## 4.5 Chop Sticks Only

The WIN32 subsystem does not have an equivalent for `fork()` or an equivalent for the `exec*()` family. There is a single primitive, named `CreateProcess()` that takes 10 arguments, yet still cannot perform the simple operation of overlaying the current process with a new program as `execve()` requires.

## 4.6 Parent/Child Relationships

The WIN32 subsystem does not support parent/child relationships between processes. The process that calls `CreateProcess()` can be thought of as the parent, but there is no way for a child to determine its parent. Most resources, such as files and processes, have handles that can be inherited by child processes and passed to unrelated processes. Any process can wait for another process to complete if it has an open handle to that process. There is a limited concept of process group that affects the distribution of keyboard signals, and a process can be placed in a new group at startup or can inherit the group of the parent process. There is no way to get or set the process group of an existing process.

## 4.7 Signals

The WIN32 API provides a structured mechanism for exception handling. Also, signals generated from within a process are supported by the API. However, signals generated by another process have no direct method of implementation. In addition to being able to interrupt processing at any point, a signal handler might perform a `longjmp` and never return.

## 4.8 Ids and Permissions

Windows NT uses *subject identifiers* to identify users and groups. A subject identifier consists of an array of numbers that identify the administrative authority and sub-authorities associated with a given user. A UNIX user or group id is a single number that uniquely identifies a user or group only within a single system. Information about users is kept in the a registry database which is accessible via the WIN32 API and the LAN manager API.

Windows NT uses an *access control list*, ACL, on each file or object to control the access of the file or object for each user. UNIX uses a set of permission bits associated with the three classes of users: the owner of the object, the group that the object belongs to, and everyone else. While it is possible to construct an access control list that more or less corresponds to a given UNIX permission, it is not always possible to represent a given access control list with UNIX permissions.

Windows NT has separate permissions for writing a file, deleting a file, and for changing the permission on a file. The write bit on UNIX systems determines all three. Thus, it is possible to encounter files that have partial write capability.

UNIX processes have real and effective user and group id's that control access to resources. Windows NT assigns each process a *security token* that defines the set of privileges that it has. UNIX systems use `setuid/setgid` to delegate privileges to processes. Windows NT uses a technique called *impersonation* to carry out commands on behalf of a given user. There is no user that has unlimited privileges as the *root* user does with UNIX. Instead the special privileges of root have been broken apart into separate privileges that can be given to one or more users. One of the biggest challenges we faced was providing the UNIX model of `setuid/setgid` on top of the WIN 32 interface.

The implementation of WIN32 for Windows 95 does not support the NT security model and calls return a *not implemented* error.

## 4.9 Terminal Interface

Windows NT and Windows 95 allow each character based application to be associated with a *console* which is similar to an `xterm` window. Consoles support echo and no echo mode, and line at a time or character at a time input mode, but lack many of the other features of the POSIX `termios` interface. There is no support for processing escape sequences that are sent to the console window. In echo mode, characters are echoed to the console when a read call is pending, not while they are typed. There are separate console handles for reading from the keyboard and writing to the screen.

## 4.10 Special Files

The WIN32 API supports unnamed pipes with the UNIX semantics. Named pipes are also supported but have different semantics than fifos and occupy a separate name space. There is no `/dev` directory to name special files such as `/dev/tty` and `/dev/null`. The WIN32 does support special names of the form `\\.\PhysicalDrive` for disk drives and tape drive devices.

Windows NT supports hard links to files, but there is no WIN32 API call to create these links. They do not support symbolic links in the file system directly, but on Windows 95 and on Windows NT 4.0, the file browser does support *short cuts* which are very similar to symbolic links.

## 4.11 Shared libraries

The WIN32 API supports the linking of shared libraries at program invocation and at run time. The libraries are called dynamically linked libraries or DLL's and are represented by two separate files,

One file provides the interface and is needed at compile time to satisfy external references. The second file contains the implementation as is needed at run time.

There are some restrictions on DLL's that are not found on UNIX system shared library implementations. One restriction is that you cannot override a function called by a DLL by providing your own version of the function. Thus, supplying your own `malloc()` and `free()` functions will not override the calls to `malloc()` and `free()` made by other DLL's. Secondly, the library can only contain pointers to data, not data itself. Thus, making a symbol such as `errno` part of a DLL is impossible. Even making symbols such as `stdin` point to data in a DLL invites trouble since it is not possible to compile code that uses

```
static FILE *myfile = stdin;
```

#### 4.12 Compilers and libraries

Microsoft sells the Visual C/C++ compiler for Windows NT and Windows 95. This compiler has both a graphical and command line interface. Microsoft also sells a software developers kit (SDK) that contains tools, including the Microsoft `rmake`. The compiler and linker use a different set of flags than standard UNIX compilers, and C files produce `.obj` files by default, rather than `.o` files. Fortunately, the linker can handle both `.obj` and `.o` files. The linker has options to choose a starting address and to specify whether the application is a console application, a GUI application, a POSIX application, or a dynamically linked library.

#### 4.13 Environment Variables

The WIN32 API supports the creation and export of environment variables in much the same way that UNIX systems do. Some environment variables, such as `PATH` are used by both WIN32 and by UNIX, yet have different formats. UNIX uses a : separated list of pathnames; WIN32 uses a ; separated list.

### 5. COMMERCIAL POSIX LIBRARY INTERFACES

We purchased software from the two commercial vendors that we were aware of that sell POSIX libraries for Windows NT that run under the WIN32 subsystem. Each offers a software development kit containing include files and libraries, and each offers a set of UNIX utilities. Both of these vendors require a license to use their libraries in products. We used earlier versions of their products but based

on their web pages at the time this paper was written, the following description still applies. Both of these vendors supply `cc` commands that invoke the underlying Microsoft Visual C/C++ compiler. Neither of these products support symbolic links, job control and fifos. Both appear to have implemented the `exec*()` family incorrectly in that the process that does the `exec` does not terminate until the child process completes. A process that repeatedly `execs` itself will eventually cause the operating system to run out of processes. It is not clear from their home pages whether either of these products work with Windows 95.

#### 5.1 NuTCracker from DataFocus

NuTCracker, by Datafocus, URL <http://www.datafocus.com>, makes an attempt to support UNIX conventions. It maps Windows NT file names to and from UNIX file names, and adjusts the `PATH` environment variable accordingly. For example, it maps the Windows NT file name `d:\bin` to the UNIX filename `/d=/bin` and handles the special names `/dev/null` and `/dev/tty`. The `=` is a poor choice because the POSIX.2 standard for the shell language leaves the behavior of commands that have an `=` in their name unspecified.

NuTCracker ships the MKS Toolkit as the utilities. The MKS Toolkit is a completely independent implementation that does not use the NuTCracker libraries. We view this as a serious deficiency since the behavior of the utilities is no guide as to the correctness or functionality of the NuTCracker library.

The NuTCracker library lacks some functions not defined by POSIX or ANSI C that are available on UNIX systems such as `hsearch()` and `cuserid()`.

In addition to the above deficiencies, NuTCracker does not support filename case distinction.

NuTCracker supports a Motif library for porting X11 based applications including a version that offers a Windows look and feel.

#### 5.2 Portage from Consensus

The other product that we purchased is named Portage and is sold by Consensus Systems, URL <http://www.consensus.com>. The source is based on System V, Release 4, which makes it the more suitable for most AT&T products. Their utilities were built from the System V source, but it was clear that changes were made in order to port

them to Windows NT.

Portage Version 1.0 does not map Windows NT file name into UNIX names. They have modified some tools such as ksh to recognize ; as the **PATH** delimiter in place of :. Version 1.0 did not support case distinction, but their home page indicates that they now do.

In terms of functionality, the NuTCracker suite is more complete than Portage.

## 6. UWIN DESIGN AND IMPLEMENTATION

We started work on writing our own POSIX library at the beginning of 1995 after being frustrated with the existing commercial products. We were able to put together a useful subset of functions in about 3 months. However, to be successful, it was necessary to provide as complete a package as possible. The library needed to handle console and serial line support, sockets, UNIX permissions, and other commonly used mechanisms such as memory mapping, IPC, and dynamic linking. In addition, to be useful, the libraries had to be documented and supported. This put the scope of the project outside of the reach of a small research department such as ours.

We subcontracted some of the development to AT&T GIS in India to help complete this project. We jointly designed the terminal interface and the group in India implemented it. They also worked on completing the sockets library. They packaged the software for installation and are providing documentation. This section describes the UWIN implementation and how we solved many of the problems described in Section 4.

### 6.1 UWIN Architecture

The current implementation of UWIN consists of two dynamically linked libraries named `posix.dll` and `ast.dll` that more or less implement the functions documented respectively in section 2 and section 3 of UNIX manuals. In addition, a server process named UMS runs as Administrator (the closest thing to root). UMS generates security tokens for `setuid/setgid` programs as needed. It also is responsible for keeping the `/etc/passwd` and `/etc/group` files consistent with the registry database. The Architecture for UWIN is illustrated in Figure 1. The UMS server does not exist for Windows 95.

The `posix.dll` library maintains an open file table that is shared by all the currently active UNIX processes in a memory mapped region. This region is writable by all processes so that an ill-behaved process could affect another process. Even though all processes have read and write access to the shared segment, secure access to kernel objects in Windows NT is not compromised by this model because a process must have access rights to an object to use it; knowing its address or value doesn't give additional access rights. Some initial measurements indicated that the alternative of having a server process update the shared memory region, would have had a performance penalty that we did not believe was worth the cost. However, this is an area for future investigation.

The open file table is an array of structures of type `Pfd_t` as illustrated in Table 1.

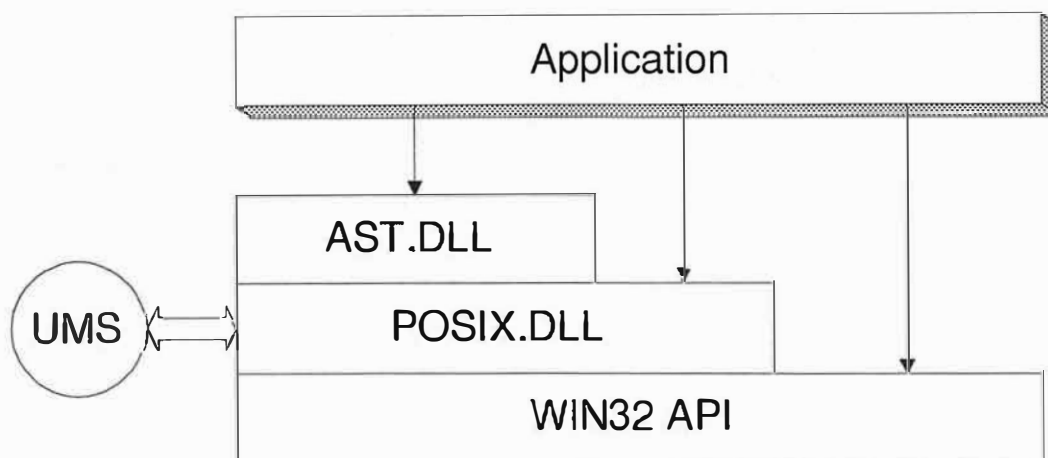
| Pfd_t |          |
|-------|----------|
| long  | refcount |
| int   | oflag    |
| char  | type     |
| short | extra    |

TABLE 1. File Table Structure

The `refcount` field is used to keep track of free entries in this table. The `Win32 InterlockedIncrement()` and `InterlockedDecrement()` functions are used to maintain this count so that concurrent access by different processes will work correctly. The `oflag` field stores the open flags for the file. The `type` field indicates what type of file, regular, pipe, socket, or special file. The function that is used read from or to write to the file depend on the value of `type`. For certain types, the `extra` field stores an index into a type-specific table that stores additional information about this file.

The `posix.dll` library also maintains a per process structure, `Pproc_t`. The per process structure contains information required by UNIX processes that is not required by Win32 processes such as parent process id, process group id, signal masks, and process state as illustrated in Table 2.

Like the open file table, the process table maintains a reference count so that process slots can be allocated without creating a critical region. The meaning of most of the fields in the process structure can be deduced by its name. The `Psig_t` structure contains the bit mask for ignored, blocked and



**Figure 1.** – UWIN Architecture

pending signals. When the first child process is invoked by a process, a thread is created that waits for this and subsequent processes to complete. The waitevent field contains an event this thread also waits on so that additional children can be added to the list of children to wait for.

| Pproc_t  |                   |
|----------|-------------------|
| long     | refcount          |
| HANDLE   | proc,thread       |
| HANDLE   | sigevent          |
| HANDLE   | waitevent         |
| HANDLE   | etok,rtok         |
| ulong    | ntpid             |
| pid_t    | pid,ppid,pgrp,sid |
| id_t     | uid,gid           |
| Psig_t   | siginfo           |
| mode_t   | umask             |
| ulong    | alarmremain       |
| int      | flags             |
| time_t   | cutime,cstime     |
| Pprocd_t | fdtab[OPEN_MAX]   |

**TABLE 2.** Process Table Structure

The process structure contains an array of up to OPEN\_MAX structures of type Pprocd\_t that is indexed by file descriptor. The Pprocd\_t structure contains the close-on-exec bit, the index of

the file in the open file table, and the corresponding handle or handles as illustrated in Table 3.

The posix.dll library implements the malloc(), realloc(), and free() interface using the Vmalloc library written by Kiem-Phong Vo<sup>[14]</sup>. The Vmalloc library provides an interface to walk over all memory segments that are allocated which is needed for the fork() implementation described later.

| Pproc_t |            |
|---------|------------|
| short   | index      |
| char    | close_exec |
| HANDLE  | primary    |
| HANDLE  | secondary  |

**TABLE 3.** Process file structure

The ast.dll library provides a portable application programming interface that is used by all of our utilities. The interface to this library is named libast.a, for compatibility with its name on UNIX systems. libast.a provides C library functions that are not present on all systems so that application code doesn't require #ifdefs to handle system dependencies. libast.a is built using the iffe command<sup>[15]</sup> to feature test the host system and determine what interfaces do not exist in the native system.

libast.a relies on the Microsoft C library for most of the ANSI-C functionality. The most significant exception to this, other than malloc() which is provided by posix.dll, is the stdio library. libast.a provides its own version of the stdio library based on calls to Sfile<sup>[16]</sup>. The Sfile library makes calls to posix.dll rather than making direct calls to the WIN32 API as the Microsoft C library does so that pathnames are correctly mapped.

The use of Sfile also provides a simple solution to the <cr><nl> problem. When a file is explicitly opened for reading as a text file, an Sfile discipline for read() and lseek() can be inserted on the stream to change all <cr><nl> sequences to <nl>. The lseek() discipline uses logical offsets so that the removal of <cr> characters is transparent. We did not provide a discipline to change <nl> to <cr><nl> since we discovered that most Windows 95 and Windows NT utilities worked without the <cr>s. The <cr>s could be inserted by a filter such as sed if required.

## 6.2 Files

The posix.dll library performs the mapping between handles and file descriptors. Usually, each file descriptor has one handle associated with it. In some cases, two handles may be associated with a file descriptor. An example of this is a console that is open for reading and writing which uses separate handles for reading and writing.

The posix.dll library handles the mapping between UNIX pathnames and WIN32 pathnames. Many UNIX programs assume that pathnames that do not begin with a / are relative pathnames. In addition, only / is recognized as a delimiter. There is only a single root directory; the operation of changing to another drive does not change the root directory. The posix.dll library maps all file names it encounters. If the file name begins with a / and the first component is a single letter, then this letter is taken as the drive letter. Thus, the UNIX filename /d/bin/date gets translated to d:\bin\date. The file name mapping routine also recognizes special file names such as /dev/tty and /dev/null. A / not followed by a drive letter is mapped to the drive that UWIN has been installed on so that programs that embed absolute pathnames for files in /bin, /tmp, /dev, and /etc work without modification.

Finally, the path search algorithm was modified to look for .exe and .bat suffixes.

One problem introduced by the pathname mapping is that passing file name arguments to native NT utilities is more difficult since it understands DOS style names, not UNIX names. A library routine was added to return a DOS name given a UNIX name.

The posix.dll library pathname mapping function also takes care of exact case matching on file systems that require it. One of the most troublesome aspects of the WIN32 API is its lack of support for pathname case distinction. It is not uncommon to have files named Makefile and makefile in the same directory in UNIX. UWIN handles case distinction by calling the WIN32 CreateFile() function both with and without the FILE\_FLAG\_POSIX\_SEMANTICS function. If they compare equal, it executes the function internally, otherwise it spawns a POSIX subsystem process to carry out the task.

## 6.3 fork/exec

The fork() system call was implemented by creating a new process with the same startup information as the current process. Before executing main(), it copies the data and stack of the parent process into itself. Handles that were closed when the new process was created are duplicated into the new process. The exec\*() family of functions was much harder to implement. The problem is that there is no way to overlay the calling process. Portage and NuTCracker have the current process wait for the child process to complete and then exit. There are two problems with this approach. First, a process that execs repeatedly will fill up the process table. More importantly, resources from the parent process are not released. Our method causes the child process to be reparented to the grandparent and the process that calls exec\*() to exit. The process id returned by the getpid() function will be the process id of the process that invoked the exec\*() function. In other cases, it will be the same process id as the WIN32 uses. To prevent that process id from being used again by WIN32, a handle to the process is kept by the grandparent process.

Even though we implemented fork() and the exec\*() family of functions, our code rarely uses them. Because the CreateProcess() function doesn't have an overlay flag, two processes need to be created in order to do both fork() and exec\*(). libast provides a spawn\*() family of functions that combines the functionality of fork()/exec\*() on systems that don't have the spawn\*() family. All functions in libast that create processes such as system() and popen()

are programmed with this interface. On most UNIX systems, the `spawn*()` family is written using `fork()` or `vfork()` and `exec*()`. We implemented `spawn*()` in our `posix.dll` library to call `CreateProcess()` directly.

## 6.4 Signals

Signals are handled by having each process run a thread that waits on an event. To send a signal to a process, the bit corresponding to the given signal number is set in the receiving process's process block, and then its signal thread event is set. The signal thread then wakes up and looks for signals. It is important for the signal handler to be executed in the primary thread of the process, since the handler may contain a `longjmp()` out of the handler function. Prior to calling `main()`, an exception filter is added to the primary thread that checks for signals. The signal thread does this by suspending the primary thread raising an exception that will activate the exception filter of the primary thread, and then resuming the primary thread.

## 6.5 Terminals

The POSIX `termios` interface is implemented by creating two threads; one for processing keyboard input events, and the other for processing output events and escape sequences. These threads are connected to the read and write file descriptors of the process by pipes. The same architecture is used for socket based terminals and serial I/O lines. Initially, these threads run in the process that created the console and make it the controlling terminal. These threads service all processes that share the controlling terminal. New threads will be created if the process that owns the threads terminates and another process is sharing the console. When a process is created, these threads are suspended and the console handles are passed down to the child. This enables a native application to run with its standard input and output as console handles. If the application has been linked with the `posix.dll`, then these threads are resumed before `main()` is called so that UNIX style terminal processing takes place. The result is that UNIX processes will echo characters as they are typed and respond to special keys specified by `stty`, whereas native WIN32 applications will only echo characters when they are read and will use Control-C as the interrupt character.

## 6.6 Ids and Permissions

Permissions for files are only available on Windows NT. Calls to get an set permissions return *not*

*implemented* errors on Windows 95. Creating a Windows NT ACL that closely corresponds to UNIX permissions isn't very difficult. The ACL needs three entries; one for owner, one for group, and one that represents the group that contains all users. Windows NT allows separate permission to delete a file and to change its security attribute. These permissions are give to the owner of a file. The UNIX `umask()` command sets the default ACL so that native applications that are run by UWIN will create files with UNIX type permissions.

Mapping of subject identifiers to and from user and group ids is more complex. UWIN maintains a table of subject identifier prefixes, and constructs the user id and group id by a combination of the index in this table and the last component of the subject identifier. The number of subject identifier prefixes that are likely to be encountered on a given machine is much smaller than the number of accounts so that this table is easier to maintain.

## 6.7 Special files and Links

Special files such as fifos and symbolic links require `stat()` information that is not kept by the NT or FAT file systems. Also, the file system does not store the `setuid` and `setgid` permission bits. With the NT file system, this extra information has been stored by using a poorly documented feature called *multiple data streams* that allows a file to have multiple individually named parts. A separate data stream is created to hold additional information about the file. The `SYSTEM` attribute is put on any file or directory that has an additional data stream so that they can be identified quickly with minimal overhead during pathname mapping.

Using multiple data streams requires the NT file system. On other file systems, fifos and symbolic links are implemented by storing the information in the file itself. The `setuid`, `setgid` functionality is not supported on these file systems.

UWIN treats Windows 95 and Windows NT 4.0 *short cuts* as if they were symbolic links. However, these links can be created with any of the UWIN interfaces. This was done by reverse engineering the format of a short cut file and finding where the pathname of the file that it referred to was stored.

Fifos are implemented by using WIN32 named pipes. A name is selected based on the creation date of the fifo file. Only the first reader and the first writer on the fifo create and connect to the named pipe. All other instances duplicate the handle of either the reader or the writer. This way all writers to a fifo

use the same handle as required by fifo semantics.

A POSIX subsystem command is also invoked to create hard links since there is no WIN32 API function to do this. Hard links fail for files in the FAT file system.

## 6.8 Sockets

Sockets are implemented as a layer on top of WINSOCK, the Microsoft API for BSD sockets. Most functions were straight forward to implement. The `select()` function proved more difficult than we had anticipated because socket handles could not be used for synchronization, and because the Microsoft `select()` call only worked with socket handles. The `posix.dll select()` function allows different types of file descriptors to be waited for.

Our first implementation of `select()` created a separate thread that used the Microsoft `select()` to wait for socket handles, and created an event for the main thread to add to the list of handles to wait for. Our second implementation used a library routine to convert input/output events on sockets to windows messages and then waited for both windows messages and handle events simultaneously. This method had the added advantages that it was possible to implement SIGIO and that it was easy to add a pseudo file device named `/dev/windows` that could be used to listen for windows messages. Adding this pseudo device made it possible to use the UNIX implementation of `tel` to port `tksh`<sup>[17]</sup> applications to Windows NT.

## 6.9 Invocation

When UWIN invokes a process, it does not know whether the process is a UWIN process or a native process. It modifies the `PATH` variable so that it uses the ; separated DOS format. It also passes open files in the same manner that the Microsoft C library does so that programs that are compiled with this library should correctly inherit open files from UWIN programs. The initialization function also sees whether a security token has been placed in its address space by the UMS server, and if so, it impersonates this token.

The POSIX library has an initialization routine that sets up file descriptors and assigns the controlling terminal starting the terminal emulation threads as required. The `posix.lib` library also supplies a `WinMain()` function that is called when the program begins. This function initializes the `stdin`, `stdout`, and `stderr` functions and then calls a

`posix.dll` function passing the address of another `posix.lib` function that actually invokes `main()`. The `posix.dll` function starts up the signal thread and sets the exception filter for signal processing as described above. The reason for this complexity is so that UNIX programs will start with the correct environment, and so that `argv[0]` will have UNIX syntax without the trailing `.exe` since many programs use `argv[0]`. Much of the complexity occurs inside the `posix.dll` part because programs do not require recompilation when changes are added there.

## 7. CURRENT STATUS

At the time of this writing, most interfaces required by the X/Open Release 4 standard have been written and work as described in the standard. The X/Open standard requires full ANSI C functionality as well. In addition, interfaces for the curses library, the sockets library, the dynamic linking library, are also working.

A C/C++ compiler wrapper has been written that calls either the Microsoft Visual C/C++ 2.x or 4.x compiler. This compiler supports the most commonly used UNIX conventions and implicitly sets default include files and libraries. In addition it has an added hook for specifying native compiler and linker options. Applications compiled with our `cc` command can be debugged with native debuggers such as the Visual C/C++ debugger. Several auto configuration programs use the output of the C preprocessor to probe the features of the system. The output format of the Microsoft C compiler caused some of the configuration programs to fail. To overcome this, a filter is inserted when running the compiler to generate preprocessor output so that existing configuration programs work. Our compiler wrapper can be invoked as `cc` for ANSI-C compilation, as `CC` for C++ compilation, and as `pcc` to build POSIX subsystem applications.

Our compiler wrapper follows the normal UNIX defaults for suffixes rather than using the Microsoft conventions; `.o`'s rather than `.obj`'s. The `.exe` suffix is not required for Windows NT since it uses permission bits to distinguish executables. However, since we also want binaries to run on Windows 95, the `.exe` suffix is added to the name of the output file if no suffix is supplied when the compiler is invoked as `cc` or `CC`.

The latest version of `ksh`, `ksh-93` was ported. The implementation supports all features of `ksh-93` including job control and dynamic linking of built-in

commands at run time. While no changes to the code should have been necessary, changes were made to `ksh` specifically for NT. The hostname mapping attribute, `typeset -H`, which has no effect on UNIX systems, was modified to call the `posix.dll` function that returns the WIN32 pathname corresponding to a given UNIX pathname. The ability to do case insensitive matching for file expansion was also added. A compile time option to allow `<cr><nl>` in place of `<nl>` was added to the shell grammar to avoid the overhead of text file processing.

About 150 UNIX tools have been ported to Windows NT, the vast majority required no changes. Common software development tools such as `yacc`, `lex`, `make` and `nmake` have also been ported. Most of the utilities are versions that we have written at AT&T over the last ten years and are easily portable to all UNIX platforms. Other utilities, such as `make`, `bc`, and `gzip` we compiled from the GNU source using `autoconfig` to generate headers and makefiles. The `yacc` and `less` utilities and the new `vi` program were ported from freely available BSD source code. In most cases, no changes were made to the original source code.

The X Windows code has two parts, the client and the server. The server had already been ported to Windows NT and Windows 95 by commercial vendors and there was no need to build UWIN version for it. In addition, the server is often running on a UNIX host. The most difficult part of porting the X Windows client code was the fact that it had `#ifdefs` for WIN32 that selected native WIN32 calls, bypassing the UWIN calls. Once this was straightened out, the compilation was straightforward.

## 8. PERFORMANCE

There are two issues to consider with respect to performance. The first is how UWIN performs compared to using the WIN32 API and/or Microsoft C library directly. The comparison of UWIN to native performance measures one of the costs involved in using UWIN as opposed to using an alternative strategy such as rewriting to the WIN32 API.

The second is how Windows NT performs relative to other UNIX systems. The performance of UNIX operating systems on the Pentium processor was investigated by Keven Lai and Mary Baker<sup>[18]</sup>, and showed that except for networking, the Linux system, URL <http://www.linux.org>, performs

the best of the UNIX systems. The comparison to a UNIX system may be important in deciding whether to choose a UNIX platform or a Windows NT platform, although other considerations often dictate this choice.

All performance comparisons were made on an Micron computer with 133MZ Pentium processor and 32M-bytes of memory. The WIN32 measurements were made using the NTFS file system on Windows NT 4.0 operating system. The UNIX measurements were made on Linux version 2.0.18 on the same hardware.

There are four sets of tests. The first set of tests, shown in Table 4, are the same ones used in the 1991 Usenix `s fio` paper. The implementation of `stdio` under UWIN uses `s fio` rather than the Microsoft implementation since the Microsoft implementation makes WIN32 calls directly rather than going through the `read()/write()` UNIX interface. The Linux tests were run with an `s fio` implementation rather than using the native implementation to make it easier to compare results. The results show that applications that are dominated by calls to `stdio` or `s fio` are likely to perform at least as well when run under UWIN.

The second set of tests, summarized in Table 5, measures the performance of certain systems calls; for example the time to open and close files, to read and write data, the time to create and delete files, and the time to open and read directories. The tests are as follows:

1. Open and close a file 10000 times.
2. Create and delete a file 10000 times.
3. Open and read a directory containing two files 10000 times.
4. Open and read a directory containing 500 files 10000 times.
5. Run `system("/bin/echo")` 100 times.

The tests were run five times and the middle three times were averaged. The first four tests report the sum of `user+system` time. The last test uses only elapsed time because of the difficulty of obtaining accumulated times for processes using `CreateProcess()` call.

These tests show that creating and deleting files in UWIN is much slower than with Linux. Much of the time difference is due to the way UWIN deletes files to provide UNIX semantics. With UNIX it is possible to delete a file while it is open, and then

create a file of the same name. Clearly a more efficient mechanism for doing this is needed.

While reading small directories is slower than with Linux, the tests show that large directories are actually faster with NT. The `system()` test shows that Linux is quite a bit faster in launching processes than NT. The NT native test, unlike the UWIN test, executes `/bin/echo` directly without running the shell so that the results are better than they might otherwise be.

The third set of benchmarks is called the Modified Andrew Benchmarks<sup>[19]</sup>. These benchmarks measure the elapsed time to perform a set of tasks such as copying files, doing recursive walks, and compiling code. The original set of Andrew Benchmarks used the native C compiler; the Modified Andrew Benchmarks come with source code for a stripped down version of `gcc` so that the differences in compilers can be eliminated. It took little effort to modify the makefiles and build the compiler. The time to run the Modified Andrew Benchmark was 110 seconds under UWIN. The time was a mere 18 seconds under Linux. This benchmark shows the effect of the slower file and process creation times. The UWIN times could be improved by using the `spawn` family of functions in place of `fork/exec` to execute the components of the compiler as the UWIN `cc` command does. In both cases the test failed to complete near the final step; creating the archive because the native archiver was unable to handle the format produced by the generated `gcc` compiler.

The final set of benchmarks that we tried to run was the benchmark suite named `lmbench`, written by Larry McVoy and presented at the 1996 USENIX conference<sup>[20]</sup>. We were able to run only a portion of these benchmarks because many of the benchmarks require the `rpc` library which hasn't been ported to UWIN yet. In addition, we omitted tests that were covered by earlier benchmarks. The results of this benchmark are presented in Table 6. While it confirms that the I/O bandwidth under UWIN is quite good, it also shows that other aspects such as pipe latency is quite large. We did not investigate this discrepancy.

## 9. FUTURE

We are in the process of running X/OPEN conformance tests on UWIN and see how close we have come to being compliant. In addition, we are trying to decide how to port the *n* dimensional file system, *n*-DFS<sup>[21]</sup>, to Windows NT. *n*-DFS provides

a mechanism to add file system services such as viewpathing and versioning. The difficulty in porting *n*-DFS is that it must also capture native WIN32 API calls to provide a transparent interface.

The current version of UWIN does not handle many of the internationalization issues well. The current implementation has been compiled for ASCII rather than UNICODE. We plan to use UTF8 encoding of UNICODE for the system call interface, and to convert to UNICODE on the NT file system. This way we do not need to build separate binaries for UNICODE.

The current version of UWIN does not support files larger than two gigabytes because the size of `off_t` is stored as a 32 bit integer. The underlying NTFS file system supports 64 bit file offsets. Since the next version of `Sfio` supports 64 bit file offsets, we plan to support large files in a future version of UWIN.

Another issue worth investigating is whether it is possible to run Linux binaries under UWIN. This would only make sense for dynamically linked programs.

Finally there are some WIN32 interfaces that could be handled through the file system interface such as the Windows NT registry and the clipboard.

## 10. CONCLUSIONS

There appear to be few if any technical reasons to move from UNIX to Windows NT. The performance of Linux exceeds that of NT 4.0 and Linux appears to be more reliable. On three occasions NT 4.0 crashed when running the performance tests. There were no crashes with Linux. However, if you want to or need to move an application to Windows 95 or Windows NT, we believe the POSIX library we developed to be superior to any of the existing commercial libraries. While in many cases the performance loss using UWIN is minimal, the performance tests show that UWIN needs improvement.

The code for the `posix.dll` library is fairly small, about 10K lines including the terminal emulator. This library runs in the WIN32 subsystem using the WIN32 API and runs under Windows 95 as well.

We hope to be able to make version 1.1 of UWIN available in binary form on the internet. Check the internet <http://www.research.att.com/sw/tools> web site for details. We hope that this will encourage

contributions of applications that have been built with UWIN.

|         |        | UWIN    |        | WIN32   |       | LINUX   |       |
|---------|--------|---------|--------|---------|-------|---------|-------|
| test    | size   | seconds | Kb/s   | seconds | Kb/s  | seconds | Kb/s  |
| fwrite  | 10000K | 0.63    | 15923  | 0.49    | 20408 | 0.85    | 11709 |
| fread   | 10000K | 0.28    | 36231  | 0.27    | 36764 | 1.03    | 9671  |
| revrd   | 10000K | 0.31    | 32679  | 0.33    | 30303 | 0.50    | 19960 |
| fw757   | 10000K | 0.76    | 13227  | 0.94    | 10593 | 1.06    | 9416  |
| fr757   | 10000K | 0.41    | 24154  | 0.36    | 27472 | 1.09    | 9199  |
| rev757  | 10000K | 0.91    | 10989  | 1.36    | 7731  | 0.66    | 15128 |
| copy&rw | 10000K | 1.01    | 9940   | 1.00    | 9999  | 1.39    | 7173  |
| seek+rw | 2000S  | 0.93    | 34566  | 0.83    | 38672 | 1.78    | 8974  |
| putc    | 5000K  | 0.87    | 5720   | 1.00    | 5020  | 2.09    | 2393  |
| getc    | 5000K  | 0.49    | 10245  | 0.50    | 9960  | 1.57    | 3194  |
| fputs   | 50000L | 0.51    | 97656  | 0.59    | 84459 | 0.88    | 57102 |
| fgets   | 50000L | 0.40    | 126262 | 0.65    | 77399 | 0.54    | 93283 |
| revgets | 50000L | 0.69    | 72254  | 2.37    | 21132 | 0.75    | 67114 |
| fprintf | 50000L | 5.84    | 8567   | 7.31    | 6838  | 5.58    | 8968  |
| fscanf  | 50000L | 4.59    | 10888  | 4.92    | 10154 | 5.60    | 8933  |

**TABLE 4.** Stdio timings

|               |       | UWIN    |      | WIN32   |      | LINUX   |       |
|---------------|-------|---------|------|---------|------|---------|-------|
| test          | count | seconds | #/s  | seconds | #/s  | seconds | #/s   |
| open/close    | 10000 | 2.88    | 3472 | 1.89    | 5291 | 0.45    | 22222 |
| create/delete | 10000 | 42.90   | 233  | 12.77   | 783  | 3.11    | 3215  |
| readdir-2     | 1000  | 9.62    | 1040 | 4.07    | 2457 | 2.80    | 3571  |
| readdir-500   | 1000  | 42.41   | 236  | 34.34   | 291  | 45.17   | 221   |
| system        | 100   | 13.62   | 7    | 5.89    | 17   | 2.84    | 35    |

**TABLE 5.** Syscall timings

| Test                   | Unit   | UWIN  | Linux |
|------------------------|--------|-------|-------|
| Null syscall           | us     | 4     | 3     |
| Pipe latency           | ms     | 295   | 35    |
| Pipe bandwidth         | MB/sec | 23.33 | 39.37 |
| File write bandwidth   | KB/sec | 1995  | 2824  |
| File read bandwidth    | MB/sec | 33.33 | 44.18 |
| Mmap read bandwidth    | MB/sec | 75.00 | 86.33 |
| Memory read bandwidth  | MB/sec | 90.91 | 82.28 |
| Memory write bandwidth | MB/sec | 76.92 | 83.26 |

**TABLE 6.** Selected *lmbench* results

## REFERENCES

1. *Practical Reusable UNIX Software*, Edited by Balanchander Krishnamurthy, John Wiley & Sons, 1995.
2. *Microsoft Win32 Programmer's Reference, Volume 2* Microsoft Press, 1993.
3. Matt Pietrek, *Windows 95 System Programming Secrets*, IDG Books, 1995.
4. *The X/Open Release 4 CAE Specification, System Interfaces and Headers*, Issue 4, Vol. 2, X/Open Co., Ltd., 1994.
5. *The OpenEdition MVS Users Guide* IBM, 1996.
6. M. Accetta et al., *Mach: A New Kernel Foundation for Unix Development*, Usenix Association Proceedings, Summer 1986.
7. Jeffrey Richter, *Advanced Windows - The Developers Guide to the Win32 API for Windows NT 3.5 and Windows 95*, Microsoft Press, 1995.
8. *POSIX - Part 1: System Application Program Interface*, IEEE Std 1003.1-1990, ISO/IEC 9945-1, 1990.
9. *POSIX - Part 2: Shell and Utilities*, IEEE Std 1003.2-1992, ISO/IEC 9945-2, IEEE, 1993.
10. Richard A. Becker, John M. Chambers, and Alan R. Wilks, *The New S Language : A Programming Environment for Data Analysis and Graphics*, Wadsworth & Brooks/Cole, New Jersey, 1988.
11. Morris Bolsky and David Korn, *The New KornShell Command and Programming Language*, Prentice Hall, 1995.
12. Glenn S. Fowler, *A Case for Make*, Software - Practice and Experience, Vol. 20, No. S1, pp. 30-46, 1990.
13. M. McKusik, W. Joy, S. Leffler, and R. Farbray, *A Fast File System for UNIX*, ACM Transactions on Computer Systems, Vol. 2, No. 3, August, 1984, 181-197.
14. Kiem-Phong Vo, *Vmalloc - A General and Efficient Memory Allocator*, Software - Practice and Experience, Vol. 26, No. 3, pp 357-374, March 1996.
15. Glenn S. Fowler, David G. Korn, John J. Snyder, and Kiem-Phong Vo, *Feature Based Portability*, Proceedings of the USENIX Symposium on Very High Level Languages, 1994.
16. David Korn and Kiem-Phong Vo, *SFIO - A Safe/Fast String/File I/O*, Proceedings of the Summer Usenix, pp. 235-255, 1991.
17. Jeffrey Korn, *Tksh: A Tcl Library for KornShell*, Fourth Annual Tcl/Tk Workshop, Monterey, CA, July 1996, pp 149-159.
18. Kevin Lai and Marry Baker, *A Performance Comparison of UNIX Operating Systems on the Pentium*, Proceedings of the San Diego Usenix, pp. 265-278, 1996.
19. John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West, *Scale and Performance in a Distributed File System*, ACM Transactions on Computer Systems, Vol. 6, No. 1, Feb 1988 pp 51-81.
20. Larry McVoy and Carl Staelin, *Imbench: Portable Tools for Performance Analysis*, Proceedings of the San Diego Usenix, pp. 279-294, 1996.
21. Glenn Fowler, David Korn and Herman Rao, "n-DFS The Multiple Dimensional File System", Trends in Software - Configuration Management, pp. 135-154, 1994.



# Protected Shared Libraries - A New Approach to Modularity and Sharing

Arindam Banerji  
Hewlett-Packard Laboratories IBM T.J. Watson Research Center  
axb@hpl.hp.com jtracey@watson.ibm.com

David L. Cohn  
University of Notre Dame  
dlc@cse.nd.edu

## ABSTRACT

Protected Shared Libraries, or PSLs, are a new type of support for modularity that form a basis for building flexible library-based operating system services. PSLs extend the familiar notion of shared libraries with protected state and data sharing across protection boundaries. Protected state information allows PSLs to be used to implement sensitive operating system services. Sharing of data across protection boundaries yields significant performance benefits. These features make PSLs a viable basis on which a complete operating system can be built largely as a set of dynamically loadable libraries without compromising protection or sacrificing performance. PSLs also allow highly flexible implementations of new functionality to be added to current commercial operating systems. A prototype PSL implementation has been built into AIX 3.2.5 and early performance results are encouraging.

## 1. Introduction<sup>1</sup>

Software flexibility relies on *modularity*, the ability to modify or replace individual software components easily. Modularity in turn relies on not only the software's internal structure, but also on the degree to which modularity is supported by the underlying operating system and the efficacy of that support. It is not surprising, therefore, that traditional monolithic systems which lack comprehensive support for modularity are characteristically inflexible and difficult to develop and maintain. Production of highly adaptable and manageable systems relies on the development of modularity support which is flexible, efficient, and easy to use.

Attempts to produce modular operating systems have generally followed one of two approaches. The first is to separate an existing operating system kernel into a microkernel that provides a basic set of fundamental constructs and one or more user-level server tasks which run on top of the microkernel and provide operating system services [Black et al. 92] [Rosier et al. 92]. This approach has been applied to a number of commercial operating systems [Batlivala et al. 92] [Borgendale et al. 94] [Golub et al. 90] [Golub et al. 93] [Malan et al. 90] [Phelan et al. 93] [Weicek et al. 93]. The second approach is to design an entirely new operating system emphasizing flexibility using object-oriented technology which generally includes language support. The second approach has primarily been relegated to academic and research environments [Bershad et al. 95] [Campbell et al. 93] [Yokote 92].

Neither of these approaches features adequate flexibility, efficiency, and ease of use. By itself the microkernel approach conveys separation of an operating system kernel along only a single line, the kernel-user boundary. Finer-grained decomposition of both the kernel-level and user-level portions remains an issue. Also, decomposition of the user-level portion into multiple user-level server tasks may be inefficient due to overhead associated with task-based protection [Condict et al. 93] [Ford & Lepreau 94] [Lepreau et al. 93] [Maeda & Bershad 93]. The language based object-oriented approach is generally applicable only to new systems.

An alternate approach to modularity which provides sufficient flexibility, efficiency, and is easily applicable to operating systems is needed. Protected Shared Libraries (PSLs) are just such an approach. PSLs extend the familiar notion of shared libraries by adding support for protected state and allowing data to be shared across protection boundaries. Protected Shared Libraries consist of two separate mechanisms: *Protected Libraries* and *Context Specific Libraries*.

Protected Libraries associate access to specific state information with each library entry point. Entrance to a library routine, which can be enacted only via a defined entry point, conveys access to data associated with the routine; access is revoked when the routine is exited. Similarly, access to a process's data segment is revoked upon entry to a Protected Library routine and restored upon returning from it. Thus, library and client data are protected from each other's code.

Context Specific Libraries, or CSLs, share a single copy of code at a single address as traditional shared libraries do, but offer significantly increased flexibility regarding shared data. They may be seen as a mechanism for encapsulating information that needs to be shared between protection domains. CSLs allow data to

---

1. David Cohn is currently on sabbatical at the IBM Corporation, Sommers, NY. This work was supported by a grant from the IBM Corporation.

be shared between clients and a service in different ways. A CSL may share a single copy of data between multiple clients, such that all clients see the same data at identical locations. Alternatively, a CSL may share data between a client and a service at a single location such that the actual contents of the shared region is associated with the calling (client) or the called (service) region. Later sections present details of the various forms of sharing supported.

The PSL infrastructure is based on two distinct hypotheses. First, given the benefits of shared libraries, it may be easier to compose user-level system services as sets of cooperating shared libraries rather than as separate processes. Associating protection with shared libraries would allow modular library-based system services to be constructed to replace traditional servers and perhaps even some privileged mode components such as loadable virtual file systems. The ease with which shared libraries can be loaded, unloaded and dynamically relocated provides flexibility not easily attained with cooperating processes. Second, sharing between cooperating entities has to be an intrinsic part of the programming model and not retrofitted through facilities such as `mmap`. CSLs allow programmers to share information in libraries and control the exact nature of sharing. This opens up possibilities for easily creating UNIX `u_block` [Goodheart & Cox 92] implementations, sharing I/O buffers across protection domains [Khalidi & Nelson 93] and sharing closures and objects across protection boundaries [Banerji et al. 94a].

The remainder of this paper proceeds as follows. The next section presents the motivation for Protected Shared Libraries, both to improve modularity and to facilitate sharing. After that, PSL semantics are described. Implementation issues are discussed in Section 4. Performance results from a prototype PSL implementation are presented in Section 5, and finally, Section 6 presents a brief discussion of the contributions made by Protected Shared Libraries.

## 2. Design Motivation

Protected Shared Libraries are motivated by two factors. First, passive protection domains, particularly shared libraries, provide an excellent basis for software modularity. Second, efficiency requires cross-domain interactions use shared data. Discussion of these observations continues in the next subsection. Following that, the overall PSL design approach is described.

### 2.1 Shared Libraries as Protection Domains

Enforced protection boundaries have been found to be a very effective software structuring tool especially for large systems [Nelson 91] [Bogle 94] [Khalidi &

Nelson 93] [Chase 94]. Protection can be enforced through a variety of means including separate address spaces [Acetta et al. 86], language support [Nelson 91], and post-processing of binary code [Wahbe 93]. Each of these approaches has been used to increase modularity and security and to facilitate debugging of large software systems. Protection has also been used to ease modification or replacement of software components [Pu 95] [Khalidi & Nelson 93] [Orr 92].

Most work regarding enforcement of protection boundaries in current operating system software, both user-level and kernel-level, has been focussed on improving the efficiency of cross-domain invocations. Invocation times have been significantly reduced by handoff scheduling [Black 89] and thread migration [Bershad 90] [Lepreau et al. 94] [Hamilton & Kougiouris 93]. These protection domains, however, have typically been associated with processes. Counter examples exist [Organick 72] [Scott et al. 90] [Wulf et al. 81], but have generally been limited to research efforts encompassing entirely new operating systems. The only system known to use passive protection domains effectively in a commercial operating system is Mach 4.0 [Lepreau et al. 94], but even that implementation is closely tied to the notion of processes. Our focus on using passive abstractions to represent protection domains is based both on the experience of others [Carter et al. 93] as well as our own [Banerji et al. 94a] that clearly demonstrate the advantages of passive modularity.

Use of passive entities, as opposed to active processes, to represent protection domains has usually taken one of two forms, objects as in Clouds and Psyche or shared libraries as in Multics.<sup>2</sup> A strong case for the support of passive objects as the basic structuring mechanism has been made [Ford 93]. The most important advantages of passive protection domains are their ability to better represent the common case of synchronous communication, their documented ability to support optimized implementations [Druschel 92] [Chase 94] [Carter et al. 93], and the ease with which they can be managed in user-level client code. The last advantage is especially important in making shared libraries a good vehicle for passive protection domains.

#### 2.1.1 Shared Library Limitations

Most commercial operating systems support shared libraries in one form or another, but semantics vary from system to system. On most UNIX systems, library code is shared, but each client process accessing a shared

2. In Multics all object modules were effectively shared libraries.

library gets its own copy of library data. This copy gets mapped into the process' private data segment and is, therefore, equally accessible by client and library code. Some systems such as OS/2 [Deitel & Kogan 92] and Windows [King 94] allow shared or *dynamically linked libraries* (DLLs) to contain shared data as well as code, but offer little or no protection. Each client task accessing a DLL has equal access to the DLL's data. Malicious or errant clients can, therefore, corrupt shared data and adversely impact other clients.

What is desired then is shared library support that features both per-client data as in UNIX, and global data as in OS/2 and Windows, but with protection. Specifically, we desire protection of library data, including per-client data, from client code, and client data from library code.

## 2.2 Cross-Domain Sharing

Increasingly, system software complexity is being addressed through use of modular protection domains. Cross-domain interactions usually take the form of fast RPC mechanisms that circumvent much of the traditional in-kernel RPC code-path. [Bershad 90] [Hamilton & Kougiouris 93] [Condict et al. 93] Efficiency of fast RPC mechanisms has been improved through use of shared message buffers [Bershad 90]. Some optimized implementations, such as the Fbufs approach [Druschel & Peterson 93], improve throughput by two orders of magnitude. Efficiency concerns, therefore, make a compelling case for sharing. Sharing is also indicated by structural considerations.

Cross-domain sharing has been used to improve structure in various parallel programming models [Scott et al. 90], and to support persistent databases [Bogle 94] [Chase 94] and shared object frameworks [Campbell et al. 93] [Banerji et al. 94]. Sharing enables cooperation between domains with limited trust [Chase 94]. Thus, sharing can be used to support a variety of interactions including producer-consumer, non-intrusive monitoring, asynchronous service providers [Bogle 94], shared pipes, stateless servers with client maintained state, and shared objects exported by servers [IBM 93].

There is a third reason for sharing across protection domains. Most implementations of cross-domain object interactions include a fair amount of overhead for the locally distributed case [IBM 93] [Janssen 95]. Thus, sharing object instances [Banerji et al. 94] and passing enclosures between protection domains on the same machine are usually inefficient. Most of these problems may be solved by judicious sharing of data and addresses between interacting protection domains. Such techniques can drastically reduce the cost of object

interaction between protection domains on the same machine.

Capturing this rich yet diverse set of structuring possibilities in an uniform abstraction requires careful design. Adequate programming support is needed so the benefits of sharing may be fully and easily exploited.

### 2.2.1 Programming Support for Sharing

Two attributes make shared information attractive to programmers.

- First, the ability to share pointer-rich data across domains is attractive. This ability has been found to be useful for persistent stores [Chase 94], shared C++ objects [Banerji et al. 94], distributed shared data and system software. The obvious argument against uniform sharing is the need to reserve portions of an address space. This concern is decreasing with the increasing popularity of large effective address spaces such as the 52-bit global address space and 64-bit non-segmented address space in the POWER [Weiss 94] and Alpha architectures respectively. The advantages in efficiency of avoiding pointer transformations in various applications and system software, as well as programmer convenience are significant.
- Second, the ability to treat shared data through symbolic names that maintain meanings across domains is attractive. A good example of this is the ability to call the shared "libc" version of malloc, uniformly from any process that links in the shared libc library. This facility can easily be extended to shared data, by involving the linker or loader in the manipulation of shared information. This approach can be seen in the shared libraries of systems as diverse as Multics, OS/2 and Hemlock [Garrett et al. 93].

Clearly, with a little system support uniform addressing and naming, can be integrated into relocatable object modules, as has been done with shared libraries in Multics, Hemlock and OS/2. In current implementations, however, the available sharing mechanisms provide limited flexibility.

Context Specific Libraries are motivated by the observation that with a few system extensions, different modes of sharing, along with uniform addressing and naming, can easily be integrated with the common notion of shared libraries. Although other efforts have provided improved forms of sharing, few have been integrated with commercial operating systems.

## 3. Protected Shared Library Semantics

Protected Shared Libraries extend traditional shared libraries in two ways, with protected state data and

cross-domain data sharing. Protected Libraries protect library and client data from each other's code. Context Specific Libraries allow global, client-specific and domain-specific data to be shared across protection domains. The semantics of each of these mechanisms is now described in detail.

### 3.1 The Mechanisms

Protected Libraries improve modularity and security and facilitate debugging of large software systems by enforcing protection domains between client and library code. Previously, other approaches to protection based on active entities such as processes have been used to improve modularity. Protected libraries investigate the alternative of using dynamically loadable passive shared libraries to enforce protection. This idea is based on efforts such as Psyche [Scott et al. 90] and Multics [Organick 72] both of which supported protected dynamically loadable object modules.

Context Specific Libraries represent modules of code and data that may be shared in various forms between different protection domains. They represent a communication channel between protection domains and thus augment traditional RPC mechanisms. CSLs extend the notion of cross-domain data and address sharing as found in Fbufs [Druschel & Peterson 93], the zero-copy I/O framework and most implementation of the UNIX u-block [Leffler et al. 89]. Together, these mechanisms form a coherent set of structuring and communication abstractions which support construction of system software at user-level.

### 3.2 Protected Libraries

Protection domain semantics are determined primarily by resource management. Traditional protection domains such as processes act as containers of resources such as memory, threads, file handles, and semaphores. With process-based protection, resource management during control transfer from one domain to another is fairly simple. Resources belonging to the currently running process are accessible. Resource management is more complex with library-based protection because resources are not typically associated with libraries.

Resources may be categorized into memory resources, such as client and service data, and non-memory resources, such as file handles and semaphores. A programmer using protected libraries typically need be concerned only with the handling of memory resources. Handling of non-memory resources is more complicated and usually only relevant to advanced programmers and library authors. Prior to describing resource management issues in detail, we define several terms.

#### 3.2.1 Definitions

Two threads executing concurrently within a protection domain always see the same set of memory resources. A UNIX process constitutes a *primary* or *root protection domain* in which execution is initiated. A Protected Library is viewed as a *secondary protection domain* which is always associated with one or more primary domains. Execution in a secondary domain is always initiated by a thread entering it from another primary or secondary domain.

A thread is the primary unit of execution and can traverse protection domains. As in most multi-threaded process models, each thread owns a small set of per-thread resources such as scheduling and accounting information. These resources, usually encapsulated in a *shuttle* [Hamilton & Kougiouris 93], belong to the thread and remain associated with the thread as it traverses protection domains. Most resources a thread accesses belong to the domain in which it is currently executing. All threads within a domain have equal access to the domain's resources. Threads could conceivably be created in either primary or secondary domains, but thread creation is currently restricted to primary domains.

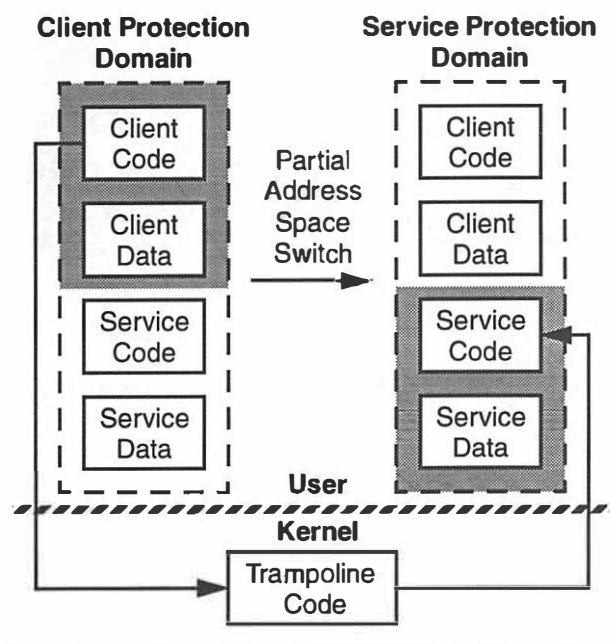
If Protected Library calls were the only form of cross-domain interaction, thread movement would be restricted to one primary domain and its associated secondary domains. Alternate mechanisms such as thread migration would allow a thread to move between multiple primary domains. The following discussion is simplified by limiting cross-domain control transfers to Protected Library invocations and eliminating consideration of other forms of IPC.

#### 3.2.2 Memory Resources

A Protected Library is an enforced unit of modularity which contains code and data. Any stateless or statefull service that requires protection from either multiple users or from other software components can be implemented in a Protected Library. Protected Libraries can be viewed as regular shared libraries that export protected entry points. Data associated with a Protected Library is accessible only once the library has been entered via a defined entry point. To summarize, a Protected Library is a passive protection domain that can be entered only through defined entry points.

Figure 1 depicts a Protected Library that does not use any Context Specific Libraries. A client process cannot access any of the service data until it calls a defined entry point. A thread starts executing in the client-domain where it can access only the process's own private code and data. This includes code and data defined in the client program and in any ordinary libraries linked

with it. Upon calling a defined service entry point, the thread enters the service protection domain. In doing so the thread loses access to its own private code and data and gains access to service code and data.



**Figure 1 Protected Library**

This description illustrates the similarity between Protected Libraries and encapsulated objects. However, Protected Libraries differ from encapsulated objects in three respects. First, Protected Libraries enforce protection boundaries. Admittedly, some object implementations also do this. Second, Protected Libraries allow for direct information sharing between a client and a service, through Context Specific Libraries as described shortly. Finally, because Protected Libraries are actually protection domains, their semantics encompass operating system and user environment resource management.

### 3.2.3 Non-Memory Resources

Protected Libraries use the “sum of resources” model to specify accessibility to non-memory resources when executing in a secondary or Protected Library domain. A thread’s resource set while executing in a Protected Library is the sum of a set of resources passed from the original primary domain, a set of resources associated with the library domain, and a subset of the per-thread resources associated with a thread no matter what domain it is executing in. The “sum of resources” model only affects Protected Shared Library (secondary) domains and not the root or primary domain which is represented by a regular process context. This follows

the Hydra [Wulf et al. 81] model of resource management which allows certain sets of resources to be passed essentially as parameters into a domain along with a call to the domain. The set of resources that must be explicitly passed when entering a secondary domain is part of a Protected Library’s interface specification.

For the most part, programmers building Protected Libraries can safely ignore the notion of passing resources from one domain to another. Default specifications allow this detail to be disregarded, in general, without complications. A programmer might have to choose a particular set of resources to be passed into a domain in the case of exception vectors, such as UNIX signal handlers, for user-level threads. Even in the case of UNIX signals, however, determination of which signals should be handled on a per-domain basis and which must be handled by the primary domain would generally be straightforward.

## 3.3 Context Specific Libraries

Context Specific Libraries (CSLs) add sharing primitives to the PSL abstraction. CSLs are units of modularity, but not protection, that may be accessible by multiple domains simultaneously. CSLs contain information that must be shared across multiple domains. The CSL is mapped into different protection domains depending on the type of sharing required. CSLs thus extend traditional cross-domain RPC with which information is shared through parameters only by adding communication via shared memory. CSLs are not protection domains but communication channels that may be associated with protection domains to share information. Depending on the type of sharing, as described below, a CSL may be viewed as a fixed piece of information accessible by all domains, mapped information that moves with a thread across domains, or in other ways. A programmer need decide only what information resides in the CSL and what kind of sharing is required. Mapping and sharing of CSLs in multiple domains is handled by the PSL implementation.

### 3.3.1 Context Specific Library Properties

A CSL is uniformly shared and named in all domains in which it is visible. This implies that symbol names and addresses seen by different domains, are consistent for a given CSL. It does not imply that all domains that use a particular CSL necessarily see the same contents. However, all domains using a particular CSL see the same resolved names at the same addresses; they may or may not see the same contents depending on the type of CSL.

As discussed in the previous section, this approach of sharing information between domains has consider-

able benefits, such as the ability to exchange pointers between domains and deal with shared data through symbolic names rather than pointers. This advantage results from encapsulating information in shared libraries. Use of shared libraries rather than a facility such as mmap or shared memory IPC implies that shared information is subject to relocation during loading. This allows uniform relocation and address space allocation required for uniform naming and sharing to be implemented relatively easily. CSLs may, therefore, be called and used from PSL based protection domains or the originating root domain with exactly the same effect. From the point of view of the calling protection domain, CSLs look like regular shared libraries with different data sharing characteristics. CSLs appear to execute in the context of the calling domain with one exception.

A CSL module executes in the kernel context of the calling domain. The kernel resources available to CSL code are, therefore, those of the client domain calling the CSL. A CSL also maintains its own user-level context. The reason for this is based on programming experience with CSLs which are frequently used to allocate and manipulate shared data. In such cases, it is extremely useful to depend on a memory allocation or malloc function that is specialized towards the kind of sharing supported by the particular CSL. Thus, a programmer requiring a particular kind of data sharing encapsulates data and its manipulating code in a CSL and uses regular malloc calls. This frees the programmer from explicitly having to deal with multiple versions of malloc.

CSLs support three distinct types of sharing each intended to support a different type of interaction between protection domains. In all three types, each protection domain accessing the CSL views the CSL at the same address. The three types of sharing differ in the contents seen by different domains.

### 3.3.2 Global Context Specific Libraries

Global CSLs are used to share data and addresses among multiple domains. A Global CSL is depicted in Figure 2. Global CSLs feature a single instance of their associated data. This instance is mapped at a single address into the address space of each primary or secondary protection domain accessing the CSL. The global sharing model relies on clients of the shared data using voluntary locking protocols to maintain coherency.

One example use of a Global CSL is for a global memory allocation facility that can be used by multiple processes and PSLs to support globally shared data. In this scenario, both addresses and their associated data must be shared. Nearly the same effect could be

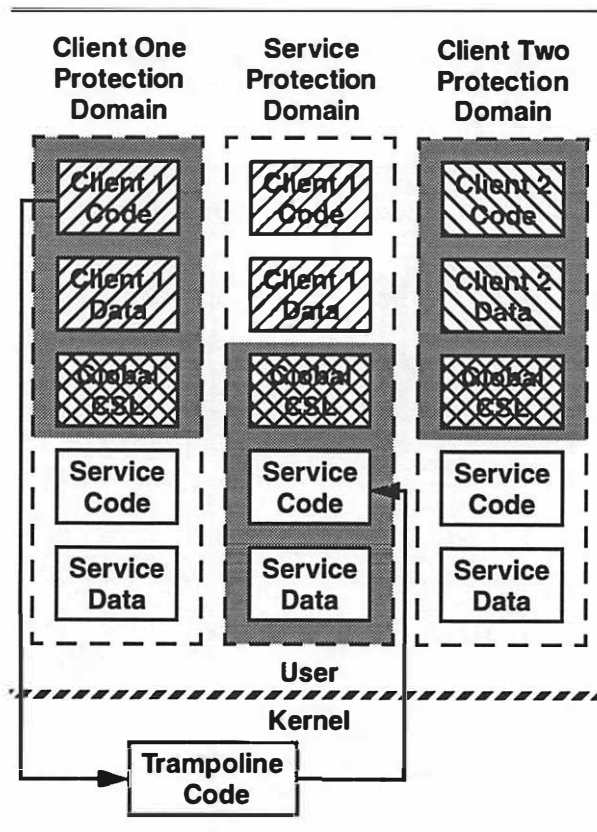


Figure 2 Global Context-Specific Library

achieved by mmaping files between different protection domains. However, the involvement of the linker in the creation of CSLs allows reference to data variables via symbolic names instead of through pointers. Global CSLs thus simplify use of shared data.

### 3.3.3 Client Context Specific Libraries

Client CSLs facilitate sharing of process-specific information between client and library domains. Shared information, encapsulated in a library, is mapped into each process's address space as shown in Figure 3. All processes get their own copy of the data, but the data is located at identical locations in all domains. Upon calling a Protected Library entry point, the CSL data of the current primary (process) domain gets re-mapped into the service domain. Thus, Protected Library code sees the CSL data belonging to the current process. Figure 3 shows that when process one calls the service, its CSL data gets mapped into the library domain. With this form of sharing, the client CSL gets mapped and unmapped as a protected call traverses multiple domains. Synchronization mechanisms based on voluntary locks may be used to ensure no two threads of the same parent process simultaneously modify shared data.

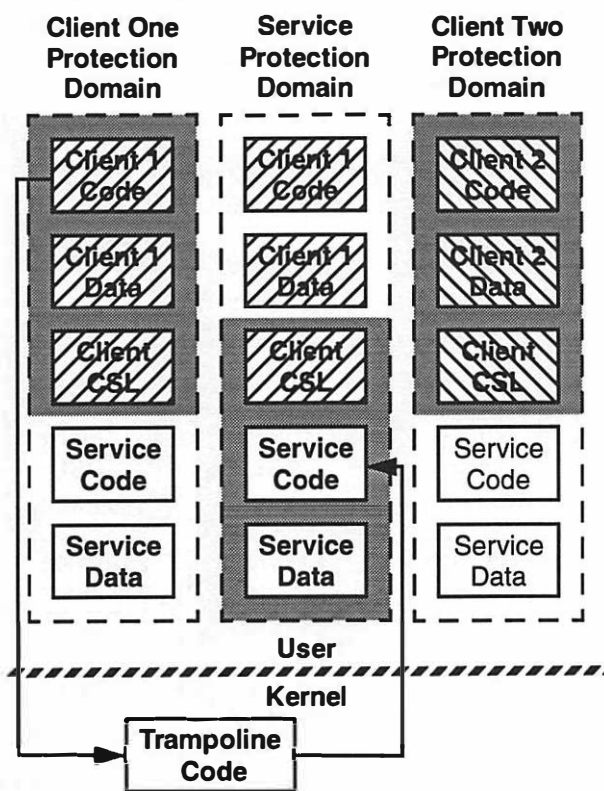


Figure 3 Client Context-Specific Library

Client CSLs can be used to implement client-specific meta-data. A Protected Library which implements a file system, for example, might maintain process specific meta-data at a certain fixed address. Whenever a thread of a particular process invokes the file system library, the library code automatically refers to the client-specific meta-data. A similar principle is used to implement u-blocks in most commercial UNIX implementations. Per-client meta-data may also be used to maintain per-client method/function tables. This allows client-process specific behavior to be automatically encapsulated within Client CSLs. Any changes to a client-specific behavior is thus easily limited to particular process.

### 3.3.4 Domain Context Specific Libraries

With a Domain Context Specific Library, depicted in Figure 4, addresses are shared, but data is not. With a Domain CSL, a distinct copy of the library data is maintained for each protection domain. The data is mapped at the same address in each domain. This form of address sharing includes both static and dynamic data. Dynamic allocation of memory in a Domain CSL may be viewed as acquisition of a shared resource, specifically the addresses shared by all clients of the Domain

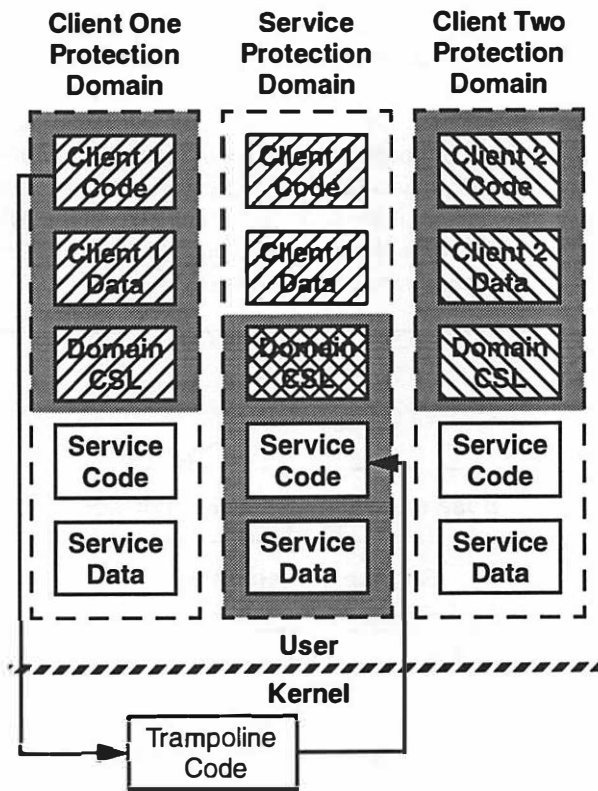


Figure 4 Domain Context-Specific Library

CSL. If a particular domain allocates memory in a Domain CSL, the allocated addresses may not be reused by any other domain. Consequently, in Domain CSLs locking is supported during address allocation but no locks are necessary when data is accessed because every domain has its own data.

One possible use of Domain CSLs is for sharing of C++ objects that contain pointers to virtual function tables (vtbls). In this case, the vtbl must be located at the same address in every domain, but its contents must be unique per domain [Banerji et al. 94]. Thus, the address of a function table can be shared while ensuring the contents of the table are unique per domain.

### 3.4 Potential Uses of PSLs

Protected Shared Libraries are a valuable tool for structuring systems. PSLs provide obvious benefits including the efficiency associated with the use of shared data and passive as opposed to active protection domains. PSLs also provide more subtle benefits as described below.

#### 3.4.1 Scope Management

Protected Libraries and CSLs may be used to efficiently implement scope management in systems that

use meta-object protocols for extensibility. Meta-object protocol based implementations offer two sets of interfaces; one provides access to normal functionality, the other optionally allows manipulation of the service implementation. The two interfaces associated with meta-object protocols are depicted in Figure 5. A critical issue in meta-object protocol implementations is ensuring that changes made to a service implementation by a client affect only the client making the changes. This issue is referred to as scope management.

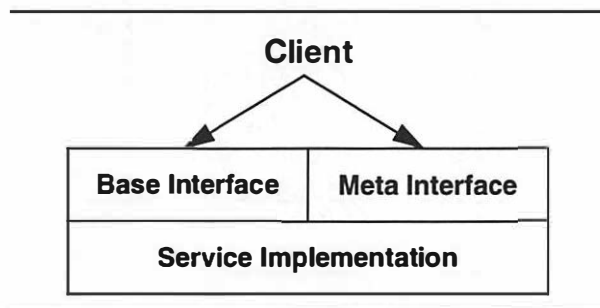


Figure 5 Meta-Object Protocol

Scope management is often implemented by maintaining per-client function or method tables. Creating these per-client method tables in client CSLs as shown in Figure 6 ensures that the service protected library always “sees” the method tables of the current calling root domain. Most UNIX implementations use a similar facility to implement `u_blocks`. However, this requires that `u_block` addresses be hard-coded and thus the functionality is limited to in-kernel co-location of `u_blocks`. Client CSLs eliminate the need for hard-coding addresses and open up the functionality to any protected library client.

### 3.4.2 Locally Distributed Objects

Distributed object implementations that cross machine boundaries need marshalling/unmarshalling and method table pointer initializations because shared memory facilities often do not extend across machines. Most marshalling/unmarshalling and method table pointer manipulations are unnecessary in distributed object implementations that do not cross machine boundaries. However, most implementations do not use shared memory to implement distributed objects efficiently in the local case [Radia 95]. Client CSLs and domain CSLs can be used to avoid marshalling/unmarshalling or method table pointer initialization into the local case.

Figure 7 shows instance data created in a client CSL with a method table in a domain CSL. The instance data gets mapped into the called domain when a locally distributed object is invoked. Use of domain CSLs

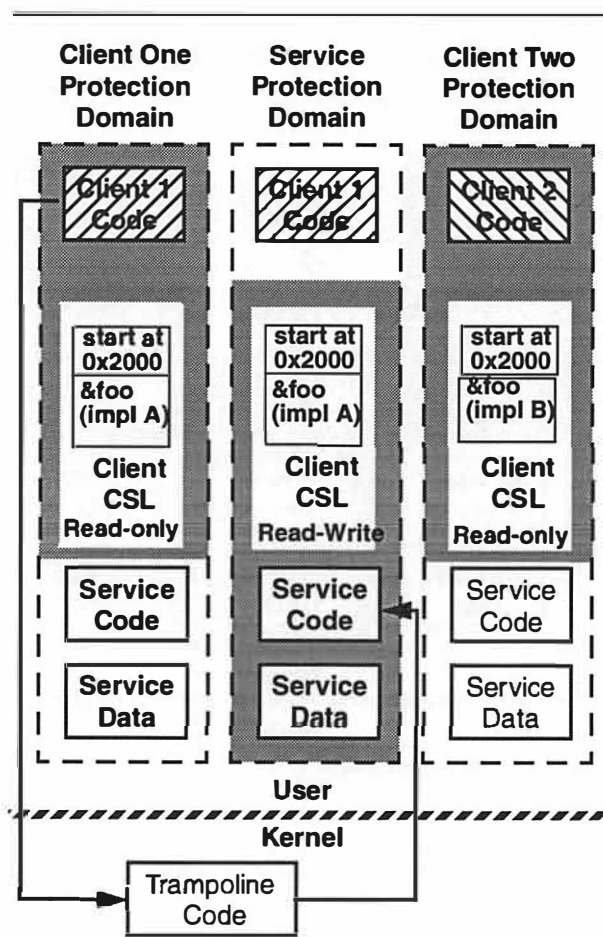


Figure 6 Scope-Management with PSLs

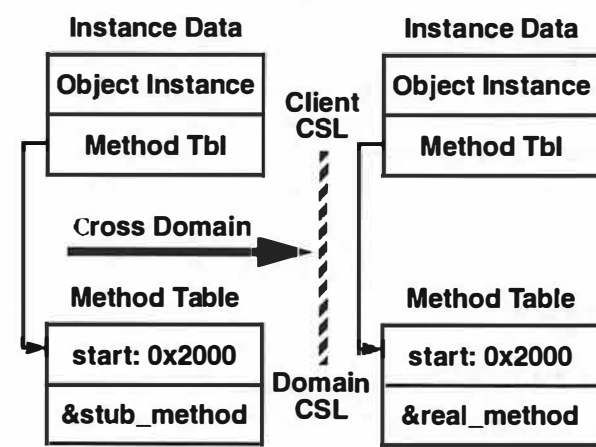


Figure 7 Locally Distributed Objects

ensures the appropriate method tables are found in each domain at identical addresses. As Figure 7 indicates the caller and callee method tables are co-located since they are created in a domain CSL, but their contents are different. In the caller, the method table contains a pointer

to a stub-method, whereas in the callee the method table contains a pointer to the actual method. Thus, judicious use of client CSLs and domain CSLs can eliminate extraneous overhead in locally distributed object implementations.

### 3.4.3 Per-Client Protection Schemes

Use of passive libraries implies per-client bindings thus allowing for flexibility in composing protected library modules. Shared libraries with or without protection require binding of client relocations to service symbols. The nature of these bindings is such that they are always maintained on a per-client basis. Thus, when shared libraries are used as protection domains, the binding between a client and a protected service may be adjusted on a per-client basis. While most processes transfer to some trampoline code to access a service entry point, a trusted client may be linked to have direct access. This allows for highly efficient trusted processes. Similarly, well-tested and debugged clients may be linked to directly access a service, bypassing protection boundaries. Finally, using facilities for dynamic binding of symbols, protection boundaries may be removed and inserted at run-time, if necessary. All these facilities, result from a single design choice, the use of shared libraries or more specifically the use of passive modules subject to relocation.

## 4. PSL Implementation

One important aspect of the PSL research to date has been the construction of a prototype implementation. The main objectives of the prototype were to clarify the PSL semantics and provide an experimental testbed for a quantitative performance analysis. The prototype was built on AIX 3.2.5, and consists of a modified AIX kernel, C runtime libraries and a new linker. This prototype is only one possible implementation of PSL semantics. Because other operating systems, executable file formats and hardware architectures may imply different implementations, only the main implementation issues are described here.

We first present an overview that draws the connection between different semantic features and aspects of the implementation. The following subsection describes the components themselves in detail. Prior to delving into implementation issues, a brief description of the RS/6000 memory architecture is presented.

### 4.1 RS/6000 Memory Architecture

A given address space on an RS/6000 is defined by a set of sixteen segment registers, each of which contains a 24-bit segment ID. The RS/6000 uses 32-bit virtual addresses, four bits of which identify a segment

which effectively extends the 32-bit virtual address to 52 bits. Of the remaining 28 bits from the original 32-bit virtual address, 16 bits identify a virtual page within the segment, and the remaining 12 bits identify a byte within the page.

By definition, addresses are not generally valid across address spaces. Regions can, however, be shared among multiple address spaces if each space loads a given segment register with the same 24-bit segment ID. As with most architectures, loading of a segment register is a privileged operation on the RS/6000.

### 4.2 Implementation Overview

PSL semantic features map rather directly to aspects of the PSL implementation. In each of the following paragraphs the relationship between a specific PSL semantic feature and the relevant aspects of the prototype implementation is described. The relationships between semantic features and implementation aspects are summarized in Table 1.

**TABLE 1 SEMANTIC FEATURES AND IMPLEMENTATION ASPECTS**

| PSL Semantics                                | Implementation Aspects   |
|--|--|
| Shared Libraries as protection domains       | Protection implemented via address space manipulation; system loader maps libraries into separate address space regions; passive domains simplify stack management |
| Partial address space switch on library call | Partial address space switch performed by trampoline code upon entering and exiting PSL routine; PSL linker replaces library calls with traps to trampoline-code   |
| Sharing between protection domains           | Shareable address space regions encapsulate shared information; trampoline-code uses architecture specific tricks to share pages between domains                   |
| Uniform addressing and naming                | Modifications to system linker (and other components) ensure addresses used to map shared data in one domain are reserved in all domains                           |

### 4.2.1 Shared Libraries as Protection Domains

The PSL implementation divides a process's address space into different regions. PSLs are mapped into these regions depending on the type of sharing and protection desired. Protection is provided by making different regions visible at any given instance. The implementation utilizes various hardware capabilities, such as page tables, segment registers and supervisor calls to control address space visibility. Division of the process-private address space into regions and mapping of libraries into these regions is the responsibility of the system loader.

### 4.2.2 Address Space Switch on Library Call

A partial address space switch is performed each time a thread enters a PSL via a protected entry point. The switch is performed by a small piece of privileged mode code called the *trampoline* code. The PSL linker ensures that threads trap to the trampoline code when calling a protected library entry point and upon returning from the PSL routine.

### 4.2.3 Sharing between Protection Domains

Sharing information between protection domains requires that certain address space regions be deemed shareable. CSLs are mapped into these shareable regions which in turn are then made visible to the appropriate domains. As a thread traverses domains, shared address space regions are mapped and unmapped as needed depending on the type of sharing. This mapping and unmapping is performed by the trampoline code.

### 4.2.4 Uniform Addressing and Naming

In order for shared data to appear at the same address in different domains, addresses which map shared data in one domain must be reserved in all domains. This reservation is ensured primarily by the system loader. However, because address allocation in UNIX is not encapsulated within the loader or any other single component, several other pieces of the kernel had to be modified to ensure reservation.

## 4.3 Implementation Components

Implementing PSLs on AIX involved modifying the AIX kernel, C run-time libraries and programming tools. We now describe the primary aspects of the PSL implementation.

### 4.3.1 Address Space Reservation

Uniform sharing requires a portion of each domain's address space be reserved. Unfortunately, in

AIX as with most UNIX implementations, address ranges are allocated independently by a number of kernel subsystems. The situation is further complicated in AIX by hard-coded starting addresses of code and data segments. To ensure address space reservation, almost all responsibility for address allocation was extracted from the various kernel subsystems and relocated to the system loader. In certain low-level assembly routines where this was not possible, address allocation logic was modified in place.

### 4.3.2 Partial Address Space Switches

The overhead associated with PSL protection is largely determined by the efficiency of the partial address space switches performed by the trampoline code. Typically an address space switch involves setting up page tables and flushing caches and translation lookaside buffers (TLBs). This process can be very costly depending on the number of pages involved and the size of the cache. Partial address space switches were implemented quite carefully in the prototype to minimize overhead and maximize performance.

Page table entries that must be switched during a domain transition are preallocated. These entries are maintained in software using a sparse representation technique [Acetta et al. 86], so a large number of pages can be represented using a small number of entries. The trampoline code only switches a couple of pointers to incorporate these new entries into the software maintained page tables. As pages are referenced in the new domain, the hardware page tables are lazily evaluated by updating them from the software tables.

Certain sets of addresses are invalidated during a partial address space switch using architecture-specific techniques. Typically, cache and TLB contents are invalidated during an address space switch to prevent cached data and translations from being used erroneously. To avoid the high cost of such invalidations, the PSL implementation implements a partial address space by changing segment register contents instead of modifying page tables. This prevents user-level threads from generating illegal addresses, while allowing for verify fast switches. Variations of this architecture-specific technique has been used on other architectures as well [Liedtke 95].

### 4.3.3 Protected Shared Library Linker

The Protected Shared Library linker subsumes the functionality of `/bin/ld`, the normal AIX linker. For binaries that are not and do not use PSLs, the PSL linker simply calls `/bin/ld`. For binaries that are PSLs or use PSLs, the PSL linker has three main responsibilities. First, wherever it detects calls to a protected library

entry point, the linker patches in a trap to trampoline code. Second, when creating a PSL, the linker adds descriptive information into unused field of the resulting binary. This information describes the kind of PSL, and type of sharing. It is used by the system loader to map the library into an appropriate address space region. Finally, the linker ensures PSL initialization and termination routines are called as needed. Like many other shared library implementations [Dietel & Kogan 92], PSLs support sub-system and per-client initialization and termination routines.

#### 4.3.4 Protected Shared Library Loader

The PSL loader replaces the AIX 3.2.5 system loader. In short, the loader implements most static aspects of PSL semantics. It creates multiple address space regions within private address spaces, maps libraries to these address space regions and generates per-domain information made available to the trampoline code. The PSL loader differs from the original AIX loader in two main ways. First, data mapping, symbol resolution and relocation were modified to ensure PSL semantics. Second, functionality was added to set up virtual memory data structures for each address space region a library is mapped into.

Typical UNIX loaders map object module data sections into a single address region, the process data segment. In contrast, the PSL loader may map object module data sections into multiple address space regions. The exact address space regions that a data section is loaded into depends on the type of PSL. This mapping and subsequent resolution and relocation creates multiple address space regions; one of which is the traditional data segment, the rest map PSL libraries as shown in Figure 2, Figure 3, and Figure 4.

The PSL loader employs a number of data structures to ensure visibility of address space regions in each protection domain. These structures include both hardware and operating system dependencies and are designed to allow path lengths through the critical trampoline code to be minimized.

#### 4.3.5 Stack Management

Passive protection domains simplify stack management during domain transitions. As in thread migration implementations [Bershad 90], there are two types of stacks. A system maintained activation stack ensures protected library calls can be nested. As a thread enters a new domain, state information for the calling domain is pushed onto the activation stack. Upon returning to the caller, the state information is restored from the activation stack and the stack is popped. The second stack is the execution stack used by almost all run-time environ-

ments. Because all secondary protection domains are passive, the execution stack of the calling thread moves with it across domains. Specifically, a thread's execution stack gets unmapped from the calling domain and mapped into the called domain during a protection domain switch. This eliminates the complexity of dynamic stack allocation associated with most thread migration implementations [Bershad 90].

#### 4.3.6 Resource Management

The current PSL prototype transfers almost all resources from the calling to the called domain during a protection domain switch. The only exceptions are signal related resources that are handled on the basis of signal type to allow for error recovery after exceptions. A complete implementation of the PSL resource handling semantics would require significant modifications of the UNIX kernel. This is a reflection on resource handling in UNIX kernels, not on PSL semantics.

Over the last decade or so, kernel code in most UNIX implementations has become quite structured. The vnode [Kleiman 86], HAT layer [Goodheart & Cox 93] and the emerging UDI interfaces [UDI 96] ensure the file-system, low-level virtual memory management services and I/O system are accessed through well-defined interfaces. This provides some degree of encapsulation and isolates clients of these interfaces from implementation changes. Furthermore, indirections such as those postulated by the stackable file systems standard [Heidemann 95] can be easily implemented. This tends to make subsystem implementations more flexible and easier to maintain.

Unfortunately the same cannot be said for process and resource management. Typically this is done through the `u_block` and `proc` structures and UNIX kernels are typically littered with direct access to these structures. Such direct access prevents any degree of encapsulation and makes it very difficult to build indirections or change UNIX resource handling. The need for encapsulation of resource handling in UNIX kernels has been previously recognized [Zajcew et al. 93]. Implementation of PSL based protection domains and process migration clearly require better encapsulation of resource handling and standardization in this area is strongly encouraged.

#### 4.3.7 Trampoline Code

The trampoline code performs the partial address space switches required when a client calls and subsequently returns from a protected shared library routine. The trampoline code performs six functions.

- Stack management
- Changing of address space visibility

- Handling of shared address space regions
- Modification of non-memory resource accessibility
- Passing of caller's resources to called domain
- Transfer of control to target entry point

The trampoline code is responsible for almost all dynamic aspects of PSL semantics. It is by far the most performance critical part of the PSL implementation. Because of this, the code is written in assembly language and pinned in memory at run time. Furthermore, in the prototype AIX implementation, the trampoline code is accessed via a special trap handler that avoids much of the overhead of a typical UNIX system call.

## 5. Performance

This section sheds light on various aspects of PSL performance. In general, Protected Shared Libraries have been found to perform better than other popular forms of cross-domain cooperation. The next subsection begins with a comparison of null-RPC times. This is followed by a comparison of PSLs with other competitive protection schemes. Finally, the section ends with a breakdown of PSL call costs. A more thorough analysis of PSL performance can be found in [Banerji 96].

### 5.1 Null-RPC Benchmark

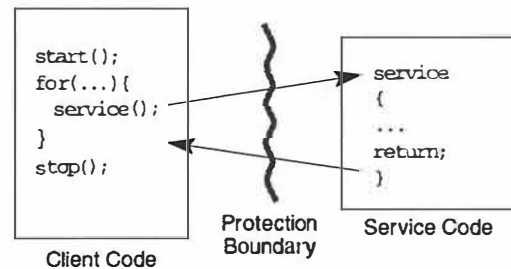
Table 2 shows the null RPC times for several RPC implementations in AIX 3.2.5 on an RS/6000 Model 530 with a relatively slow 25 Mhz POWER processor. The first two numbers are for classic user-level IPC; the third is for hand-off scheduling [Black 89]. The fourth number indicates thread migration is a bit slower than a PSL call due to resource handling overhead. The final two lines indicate the PSL null-call time is comparable to the time required for a null system call.

**TABLE 2 NULL RPC TIMES**

| Implementation                        | Null RPC Time (ms) |
|---------------------------------------|--------------------|
| User-level server task - UDP          | 2000               |
| User-level server task - System V IPC | 900                |
| User-level task - Handoff scheduling  | 190                |
| User-level task - Thread migration    | 40                 |
| Protected shared library              | 34.5               |
| System call trap                      | 22                 |

## 5.2 Benchmarks

This section evaluates five different protection



**Figure 8 Client/Service Relationship**

schemes for six different benchmark tests. In each case, the benchmark code is built as a service which is invoked by a client. The goal is to evaluate the cost of protecting the service from the client. Figure 8 shows how each invocation has to cross from client code to service code, and indicates where we start and stop our data gathering. The benchmarks used were:

- MD5, a secure one-way hash function developed to reliably identify long byte strings [Rivest 92]. The implementation used is based on code made available by RSA [RSA 93]. The input byte string is partitioned into fixed-length substrings, and the algorithm operates on the substrings in succession.
- Nsieve, a well-known benchmark that computes prime numbers. Problem size for Nsieve is the total number of primes to calculate; granularity is the number of numbers searched. The iterative portion of the Nsieve code was built as the service, so the number of invocations depends on the density of the primes.
- tdbm\_i, tdbm\_f, and tdbm\_d, three benchmarks involving the tdbm database, a small in-memory database based on the Berkeley UNIX ndbm library. This is a slight modification of the sdbm library released by Ozan Yigit [Yigit 92], and is based on the 1978 dynamic hashing algorithm by Paul Larson [Enbody 88]. Changes were made to avoid unnecessary copying and remove file dependencies. The tests involve insertion of N words from an extended version of /usr/dict/words, random fetch of N/2 words, and deletion of N/2 words.
- Nullc, a custom benchmark that is essentially a null call. The service is passed a block of data. It touches every data page and returns the data to the client. This test measures the base cost of transferring variable-sized parameters between protection

domains.

### 5.2.1 Protection Schemes

The protection schemes include both classic and

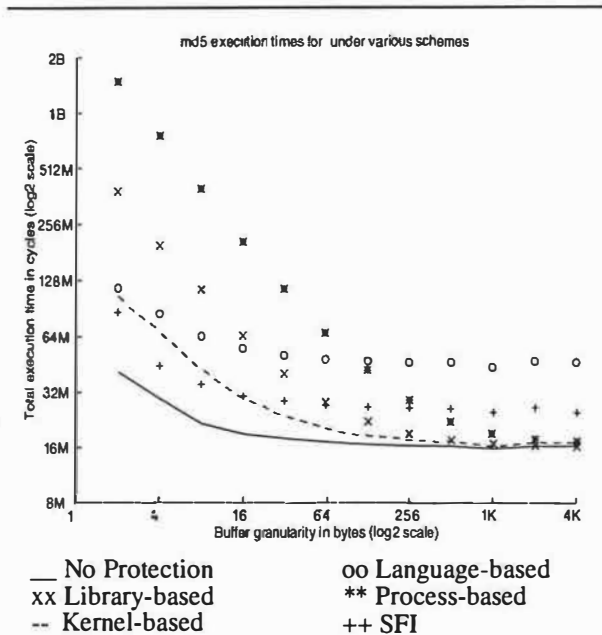


Figure 9 Execution Times for MD5

new approaches. Three are hardware based and depend upon the kernel protection boundary. The other two are software-based approaches. The protection schemes are null-protection which is used as a baseline, traditional kernel-based system calls, process-based protection with thread migration, library-based PSLs, a language-based safe-subset of Modula 3 and software-fault-isolation [Wahbe 93].

### 5.2.2 Methodology

Measurements were taken on an IBM RS/6000 Model 390 with a single 66-MHz POWER2 processor. [Weiss 94]. For each benchmark, the granularity of the protection domain was varied, and the number of machine cycles needed to perform the service was recorded. Only plots for md5 and tdbm\_d are shown here. For the md5 tests, the total message size was kept constant at 512 KBytes, and the number of bytes passed to the service with each invocation was varied. Figure 9 shows the resulting performance as a function of problem granularity. Increasing granularity increases results in fewer service invocations which causes the execution time of all schemes to decrease. For tdbm, the number of elements in the database was varied. Each invocation deals with one entry, but for larger databases the service

does more work. The results for tdbm, shown in Figure 10, differ significantly from the md5 curves in Figure 9.

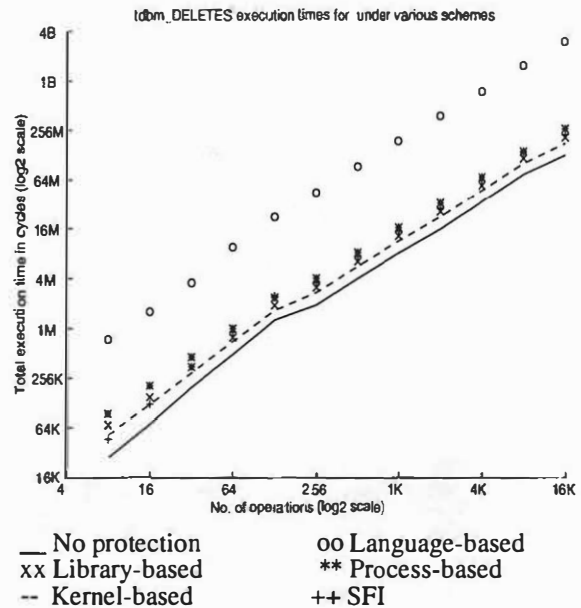


Figure 10 Execution times for tdbm\_d

### 5.2.3 Analysis

A few points about the results shown in Figure 9 and Figure 10 bear mention. First, PSLs outperformed thread migration. This is primarily due to cross-domain sharing, simplified stack management and improved resource management. Second, Figure 9 indicates that at higher problem granularities PSLs outperformed kernel based protection. With md5 this happens when the extra kernel trap and return of PSL interactions is outweighed by data copying costs of kernel interactions.<sup>3</sup> Finally, in certain cases the PSL implementation may actually outperform the unprotected case as shown in Figure 9. This is due to the PSL implementation of shared which allows data to remain in the cache between runs of different processes, whereas ordinary shared library data is always faulted into the caches on a per-process basis (at least in UNIX).

### 5.3 PSL Cost Breakdown

Figure 11 shows a breakdown of the costs associated with PSL-based protection compared to the case of no protection. Most of the overhead is due to *aliasing*,

3. With kernel trap and return times steadily decreasing (11 cycles in UltraSparc), and cache access latencies decreasing slowly, the break-even byte granularity is expected to decrease.

resolution of multiple virtual addresses to the same physical address. The POWER2 architecture provides hardware support to resolve aliases, but the support is not exploited by AIX 3.2.5.

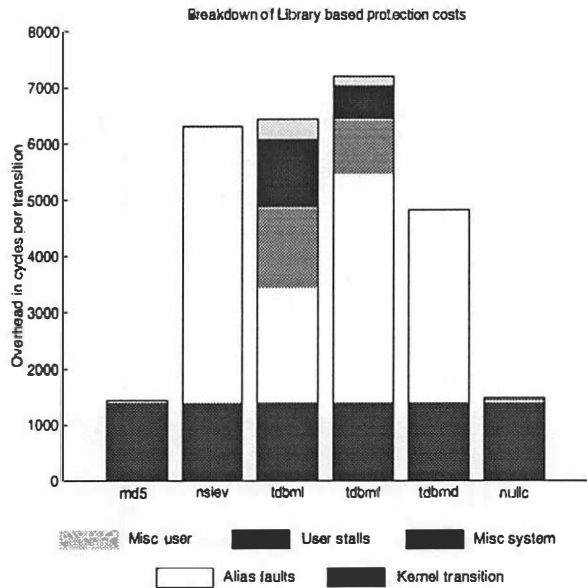


Figure 11 PSL Cost Breakdown

### 5.3.1 Aliasing Cost

Aliasing arises with hardware-based protection because client and service domains are different virtual address spaces. Consequently, virtual memory data structures must be updated when control is transferred between client and server. Also, references to shared data can cause TLB misses. The resulting *alias faults* dramatically impact transition overhead. These two costs, the in-kernel transition cost of updating aliasing data structures and extra faults due to aliasing, significantly impact PSL performance.

Figure 11 indicates the in-kernel transition cost, which includes adjustments to aliasing data structure, is essentially the same for all benchmarks. For *nullc* and *md5*, only one domain actually touches the shared data, so there are no alias faults. For the other tests, however, both client and library code access the data which results in considerable time spent handling aliasing faults. Excluding time for alias faults, for five of the six tests, the PSL overhead lies between 1300 and 2500 cycles.

To assess the cost of updating aliasing data structures, the number of instructions in the transition code was counted for the simplest service, *nullc*, with a four byte transfer. This code calls AIX routines, written in C, which adjust the virtual memory data structures. Thus,

the difference between the transition code instruction count (210) and the total instructions used to make the transition (1250) provides an indication of the cost of aliasing (1040 instructions approx). Hence, if we ignore the costs due to aliasing, which can be safely done for most modern hardware, the cost of a PSL call is actually the cost of executing 210 instructions and the kernel trap/return times. This cost is 275 cycles for the 66 MHz IBM POWER2.

### 5.3.2 Trap costs

Given that the aliasing problem can be solved in hardware, it is important to look at the other costs. These are approximately 210 instructions per transition, or about 275 cycles including kernel trap and return. This cost approaches those for highly optimized cross-domain transfer mechanisms [Hamilton & Kougiouris 93].

Without aliasing, the hardware cost of the kernel trap and return is significant, 57 cycles for the POWER2. Library-based protection traps twice for every service invocation, thus 114 of the remaining 275 overhead cycles are due to the trap and return. Some modern processors, such as the UltraSparc, have reduced trap overhead to as little as 11 cycles (that is, 22 cycles for PSL-like double trap and returns) and trap costs are expected to continue decreasing.

## 6. Discussion

Shared libraries form an excellent basis of modularity for structuring large systems. Protected Shared Libraries enhance the popular notion of shared libraries in two ways, by adding protection and allowing data to be shared across protection boundaries. This enables PSLs to be used to securely implement sensitive services. Sharing reduces many of the costs of cross-domain interactions, thus making it a viable alternative to language-based or process-based protection schemes. A prototype PSL implementation has demonstrated the efficiency of the PSL approach.

PSLs are well-suited for use in large user-level applications and for implementation of operating system services at user level as is common in microkernels. There are also less-obvious uses for PSLs. First, dynamically loadable kernel extensions suffer from their ability to corrupt kernel data. Extensions could be prevented from corrupting data to which they do not need write access by building them as dynamically loadable privileged mode PSLs. Thus, PSLs may provide safe ways of extending existing operating system kernels. Second, one important research area in operating systems is the design of low-level nanokernels or exokernels [Engler 95]. These kernels provide low-level protected inter-

faces to the hardware with most operating system functionality implemented in library routines. PSLs are an attractive approach to protecting such operating systems from application code.

## References

[Acetta et al. 86] Acetta, M., Baron, R., Golub, D., Rashid, R., Tevanian, A. and Young, M. "Mach: A New Kernel Foundation for UNIX Development." In *Proceedings of the Summer 1986 USENIX Conference* (Atlanta, GA, July). The USENIX Association, Berkeley, CA, 1986, pp. 93-112.

[Banerji et al. 94] Banerji, A. et al. "Shared Objects and vtbl Placement - Revisited." *Journal of C Language Translation*, September 1994, pp. 31-46.

[Banerji et al. 94a] Banerji, A., Kulkarni, D. and Cohn, D. "A Framework for Building Extensible Class Libraries." In *Proceedings of the 1994 USENIX C++ Conference*, 1994, pp. 26-41.

[Banerji 96] Banerji, A. et al. "Quantitative Analysis of Protection Options." Technical Report (unnumbered), University of Notre Dame, Notre Dame, IN, 1996.

[Battivala et al. 92] Battivala, N., Gleeson, B., Hamrick, J., Lurndal, S., Price, D., Soddy, J. and Abrossimov, V. "Experience with SVR4 Over CHO-RUS." In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures* (Seattle, WA, April 27, 28). The USENIX Association, Berkeley, CA, 1992, pp. 223-241.

[Bershad, 90] Bershad, B. "Lightweight Remote Procedure Call." *ACM Transactions on Computer Systems*, \*(1), February, 1990.

[Bershad et al. 95] Bershad, B. N., Saveag, A., Pardyak, P., Sirer, E. G., Fiuczynski, M. E., Becker, D., Chambers, C. and Eggers, S. "Extensibility, Safety and Performance in the SPIN Operating System." In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles* (Copper Mountain Resort, CO, Dec. 3-6). ACM Press, NY, 1995, pp. 267-284. (<http://www.cs.washington.edu/research/projects/spin>)

[Black 89] Black, D. "Scheduling support for Concurrency and Parallelism in the Mach Operating System." Unpublished.

[Black et al. 92] Black, D. et al. "Microkernel Operating System Architecture and Mach." In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures* (Seattle, WA, April 27, 28). The USENIX Association, Berkeley, CA, 1992, pp. 11-30.

[Bogle 94] Bogle, P., and Liskov, B. "Reducing Cross Domain Call Overhead Using Batched Futures." In *Proceedings of OOPSLA 94*, ACM, 1994.

[Borgendale et al. 94] Borgendale, K., Bramnick, A. and Holland, I. M. "Workplace OS: What is the OS/2 Personality?" March 24, 1994.

[Campbell et al. 93] Campbell, R. et al. "Designing and Implementing Choices: An Object-Oriented System in C++." *Communications of the ACM*, 36(9), 1993, pp. 117-126.

[Carter et al. 93] Carter, J. et al. "FLEX: A Tool for Building Efficient and Flexible Systems." In *Proceedings of the Fourth Workshop on Workstation Operating Systems* (Napa, CA, Oct. 14, 15). IEEE Computer Society Press, Los Alamitos, CA, 1993, pp. 198-202.

[Chase 94] Chase, J., et al. "Sharing and Protection in a Single Address Space Operating System." *ACM Transactions on Computer Systems*, Vol. 12(4), November 1994, pp. 271-307.

[Condict et al. 93] Condict, M., Mitchell, D. and Reynolds, F. "Optimizing Performance of Mach-based Systems By Server Co-Location: A Detailed Design." August 10, 1993.

[Deitel & Kogan 92] Deitel, H. M. and Kogan, M. S. *The Design of OS/2*. New York: Addison Wesley, 1992.

[Druschel 92] P. Druschel et. al. "Beyond Microkernel Design: Decoupling Modularity and Protection in Lipto." In *Proceedings of the 12th International Conf. on Distributed Computing Systems*, IEEE Computer Society Press, Los Alamitos, CA, pp. 512-520.

[Druschel & Peterson 93] Druschel, P. and Peterson, L. L. "Fbufs: A High-Bandwidth Cross-Domain Transfer Facility." In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles* (Asheville, NC, December 5-8). ACM Press, New York, NY, 1993, pp. 189-202.

[Enbody 88] Enbody, R. and Du, H. "Dynamic Hashing Schemes." *ACM Computing Surveys*, Vol. 20, No. 2, 1988, pp. 85-113.

[Engler 95] Engler D., et al. "Exokernel: An Operating System Architecture for Application-Level Resource Management" In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (Copper Mountain Resort, CO, Dec. 3-6), December 1995.

[Ford & Lepreau 94] Ford, B. and Lepreau, J. "Evolving Mach 3.0 to a Migrating Thread Model." In *Proceedings of the Winter 1994 USENIX Technical Conference* (San Francisco, CA. Jan. 17-21). USENIX Association, Berkeley, CA, 1994, pp. 97-114.

[Garrett et al. 93] Garrett, W. E., et al. "Linking Shared Segments." In *Proceedings of the Winter 1993 USENIX Conference* (San Diego, CA, January 25-29), The USENIX Association, Berkeley, CA, 1993, pp. 13-27.

[Golub et al. 90] Golub, D., Dean, R., Florin, A. and Rashid, R. "Unix as an Application Program." In *Proceedings of the Summer 1990 USENIX Conference* (Anaheim, CA. June 11-15). The USENIX Association, Berkeley, CA, 1990, pp. 87-95.

[Golub et al. 93] Golub, D. B., Manikundalam, R. and Rawson, F. L. III. "MVM - An Environment for Running Multiple Dos, Windows and DPMI Programs on the Microkernel." In *Proceedings of the Third USENIX Mach Symposium* (Santa Fe, NM. April 19-21). USENIX Association, Berkeley, CA, 1993, pp. 173-190.

[Goodheart & Cox 93] Goodheart, B. and Cox, J. *The Magic Garden Explained*. New York: Prentice Hall, 1993. (ISBN 0-13-098138-9)

[Hamilton & Kougiouris 93] Hamilton, G. and Kougiouris, P. "The Spring Nucleus: A Microkernel for Objects." In *Proceedings of the Summer 1993 USENIX Conference* (Cincinnati, OH, June). The USENIX Association, Berkeley, CA, 1993.

[Heidemann 95] Heidemann, J. S. *Stackable Design of File Systems*. Ph.D. Dissertation, University of California, Los Angeles, 1995.

[IBM 93] SOMObjects Developer Toolkit User's Guide, Version 2.0, June 1993, IBM, Austin, TX.

[Janssen 95] Janssen, B., et al., ILU 1.7 Reference Manual, Xerox Corporation, January 1995.

[Khalidi & Nelson 93] Khalidi, Y. A., and Nelson, M. N. "An Implementation of Unix on an Object-Oriented Operating System." In *Proceedings of the Winter 1993 USENIX Conference*. The USENIX Association, Berkeley, CA, 1993, pp. 469-479.

[King 94] King, A. *Inside Windows 95*. Redmond, WA: Microsoft Press, 1994.

[Kleiman 86] Kleiman S. "Vnodes: An Architecture for Multiple File System Types in Sun UNIX." In *Proceedings of the Summer 1986 USENIX Conference*, June 1986, pp. 238-247.

[Leffler et al. 89] Leffler, S., McKusick, M. K., Karels, M. J. and Quarterman, J. S. *The Design and Implementation of the 4.3 BSD UNIX Operating System*. New York: Addison-Wesley Publishing Company, 1989. (ISBN 0-201-06196-1)

[Lepreau et al. 93] Lepreau, J. et al. "In\_Kernel Servers on Mach 3.0: Implementation and Performance." In *Proceedings of the Third USENIX Mach Symposium* (Santa Fe, NM. April 19-21). USENIX Association, Berkeley, CA, 1993, pp. 39-55.

[Lepreau et al. 94] Lepreau, J., et. al. "The Flux Operating System Project." <http://www.cs.utah.edu/projects/flexmach>.

[Liedtke 95] Liedtke, J. "On  $\mu$ -Kernel Construction." In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles* (Copper Mountain Resort, CO. Dec. 3-6). ACM Press, New York, NY, 1995, pp. 237-250.

[Maeda & Bershad 93] Maeda, C. and Bershad, B. N. "Services without Servers." In *Proceedings of the Fourth Workshop on Workstation Operating Systems* (Napa, CA. Oct. 14, 15). IEEE Computer Society Press, Los Alamitos, CA, 1994, pp. 170-176.

[Malan et al. 90] Malan, G., Rashid, R., Golub, D., and Baron, R. "DOS as a Mach 3.0 Application." In *Proceedings of the USENIX Mach Workshop* (Burlington, VT. Oct.). The USENIX Association, Berkeley, CA, 1990, pp. 27-40.

[Nelson, 91] Nelson, G., *Systems Programming with Modula-3*. Englewood Cliffs, NJ: Prentice Hall, 1991.

[Orr 92] Orr, D. and Mecklenburg, R. W. "OMOS - An Object Server for Program Execution." In *Proceedings of the International Workshop on Object Oriented Operating Systems*, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 200-209.

[Organick 72] Organick E., *The Multics System: An Examination of its Structure*, Cambridge: The MIT Press, 1972.

[Phelan et al. 93] Phelan, J. M., Arendt, J. W., and Ormsby, G. R. "An OS/2 Personality on Mach." In *Proceedings of the Third USENIX Mach Symposium* (Santa Fe, NM. April 19-21). The USENIX Association, Berkeley, CA, 1993, pp. 191-201.

[Pu 95] Pu, C., et al. "Optimistic Incremental Specialization: Streamlining a Commercial Operating System." In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles* (Copper Mountain Resort, CO. Dec. 3-6). ACM Press, New York, NY, 1995.

[Radia 95] Radia S., et al, The Spring Object Model, *Proceedings of the Conference on Object Technologies and Systems*, July 1995.

[Rivest 92] Rivest, R. The MD5 Message-Digest Algorithm, *Network Working Group RFC 1321*, 1992.

[RSA 93] <http://www.rsa.com/pub/md5.txt>

[Rosier et al. 92] Rosier, M., Abrossimov, F., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrman, F., Kaiser, C., Langlois, S., Léonard, P., and Neuhauser, W. "Overview of the Chorus Distributed Operating System." In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures* (Seattle, WA. April 27, 28). The USENIX Association, Berkeley, CA, 1992, pp. 39-69.

[Scott et al. 90] Scott, M. L., LeBlanc, T. J., and Marsh, B. D. "Multi-Model Parallel Programming in Psyche" In *Proceedings of the Second ACM Symposium on Principles and Practice of Parallel Programming* (Seattle, WA, March 14-16), 1990, pp. 70-78.

[UDI 96] Uniform Driver Interface, [ftp://tel-ford.nsa.hp.com/pub/hp\\_stds/udi/home.html](ftp://tel-ford.nsa.hp.com/pub/hp_stds/udi/home.html)

[Wahbe 93] Wahbe, R., et. al. "Efficient Software-based Fault Isolation." In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, December 1993, pp. 203-216.

[Weiss 94] Weiss, S., Smith, J., *POWER and PowerPC*, San Francisco: Morgan Kauffman Publishers, Inc., 1994.

[Wiecek et al. 93] Wiecek, C. A., Kaler, C. G., Fiorelli, S., Davenport, W. C. Jr., and Chen, R. C. "A Model and Prototype of VMS Using the Mach 3.0 Kernel." In *Proceedings of the USENIX Symposium on Microkernels and Other Kernel Architectures* (Seattle, WA. April 27, 28). The USENIX Association, Berkeley, CA, 1992, pp. 187-203.

[Wulf et al. 81] Wulf, W. A., Levin, R. and Harbison, S. P. *Hydra/C.mmp: An Experimental Computer System*, McGraw-Hill, New York, 1981.

[Yigit 92] Yigit, O. <ftp://ftp.x.org/contrib/util/sdbm>

[Yokote 92] Yokote, Y. "The Apertos Reflective Operating System: The Concept and its Implementation." In *Proceedings of the Seventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '92)*, ACM Press, NY, 1992, pp. 414-434.

[Zajcew et al. 93] Zajcew, R. et al. "An OSF/1 UNIX for Massively Parallel Multicomputers." In *Proceedings of the 1993 Winter USENIX Conference*, The USENIX Association, Berkeley, CA, 1993, pp. 449-468.



# Extending the Operating System at the User Level: the Ufo Global File System \*

Albert D. Alexandrov, Maximilian Ibel, Klaus E. Schauser, and Chris J. Scheiman

*Department of Computer Science  
University of California, Santa Barbara  
Santa Barbara, CA 93106  
{berto,ibel,schauser,chriss}@cs.ucsb.edu  
<http://www.cs.ucsb.edu/research/ufo>*

## Abstract

In this paper we show how to extend the functionality of standard operating systems completely at the user level. Our approach works by intercepting selected system calls at the user level, using tracing facilities such as the `/proc` file system provided by many Unix operating systems. The behavior of some intercepted system calls is then modified to implement new functionality. This approach does not require any re-linking or re-compilation of existing applications. In fact, the extensions can even be dynamically “installed” into already running processes. The extensions work completely at the user level and install without system administrator assistance.

We used this approach to implement a global file system, called Ufo, which allows users to treat remote files exactly as if they were local. Currently, Ufo supports file access through the FTP and HTTP protocols and allows new protocols to be plugged in. While several other projects have implemented global file system abstractions, they all require either changes to the operating system or modifications to standard libraries. The paper gives a detailed performance analysis of our approach to extending the OS and establishes that Ufo introduces acceptable overhead for common applications even though intercepting system calls incurs a high cost.

**Keywords:** operating systems, user-level extensions, `/proc` file system, global file system, global name space, file caching

\*This work was supported by the National Science Foundation under NSF CAREER Award CCR-9502661 and NSF Postdoctoral Award ASC-9504291. Computational resources were provided by the NSF Instrumentation Grant CDA-9529418 and Sun Microsystems. The software is available on-line under <http://www.cs.ucsb.edu/research/ufo>

## 1 Introduction

Computer users have always had the desire to extend operating systems functionality to support new protocols or meet new usage patterns. In this paper we show how to extend a standard Unix operating system (Solaris) completely at the user level. Our approach — which is similar to interposition agents [Jon93] — uses tracing facilities to intercept selected system calls at the user level. The behavior of intercepted system calls is then modified to implement new functionality. We use this to implement Ufo,<sup>1</sup> a global file system, which supports file access through the FTP and HTTP protocols and can easily be expanded to support new protocols.

While this paper focuses on extending the file system services, our approach provides a general way of expanding operating system functionality without any kernel changes or library modifications. Our extensions, which are not just limited to Solaris, do not require any re-linking or re-compilation of existing applications. In fact, they can even be dynamically “installed” into already running processes. Extensions can be added on a per-user basis, i.e., extensions for one user do not affect other users. Actually, even a single user could run different jobs with different extensions without interference.

An important advantage of this method is that developing the OS extensions can be done entirely at the user level and without access to OS source code. This makes our approach an excellent way for testing new kernel extensions and for providing OS extensions that are not performance critical.

### 1.1 Personalized Global File Systems

With the recent explosive growth of the Internet an increasing number of users, including us, have access to

<sup>1</sup>The acronym Ufo stands for User-level File Organizer.

multiple computers that are geographically distributed. The initial motivation for our work was the desire to have transparent file access from our Unix machines to our personal accounts at remote sites. In addition, we also wanted to present the resources from the large number of existing HTTP and anonymous FTP servers as if they were local files. This would allow all local applications to transparently access remote files.

Ufo implements a global file system that provides this functionality. It is a user-level process that runs on multi-user Unix systems and connects to remote machines via authenticated and anonymous FTP and HTTP protocols. It provides read and write caching with a weak cache consistency policy.

It was important to us that the file system not only run at the user-level but that it also be user-installable (i.e. installing it does not require root access). For example, assume one of us obtains a new account at an NSF supercomputer center. Once we log into that account, we would like to transparently see all remote files we have some way of accessing (be it via telnet, FTP, rlogin, NFS, or HTTP); all without having to ask the system administrator to install anything. This is not necessarily easy to do in a Unix environment since most current file system software must be installed by a system administrator. For example, systems such as NFS and AFS allow sharing of files across the Internet, but they require root access to mount or export new file partitions. The system administrator may not have the time or, due to security concerns, may not be willing to install a new piece of software or export a file system resource.

A user-installable file system does not have these problems. Not only can users install it themselves, but it does not introduce any additional security holes in the underlying operating system or network protocol. To guarantee that a file system can indeed be installed by the user, it should only rely on functionality provided by standard (unmodified) operating systems.

## 1.2 Personalizing the Operating System: The Ufo Approach

In order to provide a global file system we need extensions to the operating system that handle file accesses (and related functions) properly. By modifying the behavior of the system calls, we can add new functionality to the operating system. In our approach we modify the system call behavior by inserting a user-level layer, the *Catcher*, between the application and the operating system.

The *Catcher* is a user-level process which attaches to an application and intercepts selected system calls issued by the application. From the user's perspective, the *Catcher* provides a user-level layer between the user's application processes and the original operating system,

as shown in Figure 1. This extra layer does not change the existing OS, but allows us to control the user's environment, either by modifying function parameters, or issuing additional service requests.

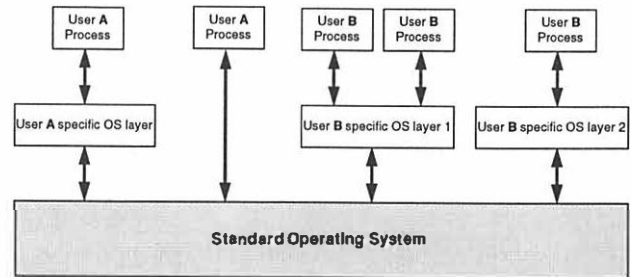


Figure 1: A new view of the operating system.

The *Catcher* operates as follows: Initially, it connects to the user process and tells the operating system which system calls to intercept. Our implementation which runs under Solaris 2.5.1 uses the System V /proc interface,<sup>2</sup> which was originally developed for debugging purposes [FG91]. Instead of just tracing the system calls, we actually change at the user-level the semantics of some of them to implement the global file system. Whenever a system call of interest begins (or completes), the operating system stops the subject process and notifies the *Catcher*. The *Catcher* calls the appropriate extension function, if needed, and then resumes the system call.

For our global file system, we intercept the *open*, *close*, *stat*, and other system calls that operate on files. When we intercept a system call which accesses a remote file, we first ensure that an up-to-date copy is available locally. Then, we patch the system call to refer to the local copy and allow it to proceed. System calls which only access local files are not modified (they can just continue), while most systems calls not related to files are not even intercepted. Since no application binaries are changed, this approach works transparently with any existing executable (with the exception of the few programs requiring *setuid*).

A potential concern with our approach is its performance overhead. While the cost for intercepting system calls is significant, our performance analysis shows that Ufo introduces acceptable overhead for common applications.

Since Ufo runs fully at the user-level, if one user runs it there is no performance penalty on another user. Furthermore, a user can run Ufo only on some selected applications without impacting other applications,

<sup>2</sup> Similar functionality is provided by Digital Unix, IRIX, BSD and Linux. This mechanism is used by system call tracing applications such as *truss* or *strace*.

or even dynamically “install” (attach) or “uninstall” (detach) Ufo while applications are running.

### 1.3 Using Ufo

Installing Ufo can be done by any user without needing root assistance. The simplest way to start using Ufo is to explicitly start processes under its control, e.g.

```
tcsh% ufo csh
csh% grep UCSB http://www.cs.ucsb.edu/index.html
csh% cd /ftp/schauser@cheetah.cs.ucsb.edu/
csh% emacs papers/ufo/introduction.tex &
```

In the example above, the new shell running under Ufo can use the global file system's services. Ufo automatically attaches to any child that the shell spawns, like the grep and emacs processes above. Alternatively, Ufo can be instructed to dynamically attach to an already running process by providing its pid.

```
tcsh% emacs &
[3] 728
tcsh% ufo -pid 728
```

### 1.4 Outline

The remainder of the paper is structured as follows. Section 2 reviews related work and compares our method for operating system extension with alternative approaches. Section 3 describes in detail the Catcher and how it intercepts system calls at the user-level. Section 4 discusses the design decisions of Ufo, the user-level global file system. Section 5 presents experimental results for a variety of micro-benchmarks, standard Unix file system benchmarks, and full application programs. Section 6 concludes this paper and offers an outlook on future research directions.

## 2 Related Work

Before presenting implementation details of the Catcher (in Section 3), we will put our work in context by comparing our approach with alternative ways of extending operating system functionality. The eager reader can skip directly to the discussion of our implementation in Section 3. We first introduce a classification of different approaches to extending the operating system. We then discuss the relevant research projects on extending operating system and file system functionality in more detail.

### 2.1 Approaches for Extending the Operating System

There has been a considerable amount of work on extending operating systems with new functionality. We

can classify these approaches into the following categories:

**Change Operating System:** The most straightforward approach is to just modify the operating system itself and incorporate the desired functionality. This requires access to the OS sources and the privileges to install the new kernel.

**Device Driver:** Instead of changing the kernel itself, modifications can be limited to a new device driver which implements the desired functionality. Root access is required to install the device drivers.

**Network Server:** A clean solution with minimal intrusion to the operating system is to install a network server, which provides the additional services through an already existing standardized interface. Installing the server and mounting remote directories requires root capabilities.

We want to re-iterate, these first three approaches require super-user intervention and affect everybody using the system, since everybody will see the modifications to the operating system. If there is a bug or security hole in the newly installed software, the whole system's integrity and security can be compromised. A user-level approach avoids this problem.

**User-level Plug-Ins:** When a one-time modification to the operating system can be tolerated, a flexible strategy is to add hooks to the operating system so that system calls can trigger additional functions that extend the functionality. This approach is especially appropriate if the OS has already been designed to be flexible and support extensions.

#### User-level Libraries (Static or Dynamic Linking):

Most applications do not directly access the operating system, but use library functions embedded in standard libraries. Instead of modifying all binaries or the OS kernel, it suffices to make changes to the libraries. Super-user privileges are only necessary if the original libraries/binaries need to be replaced.

**Application Specific Modifications:** Instead of incorporating the modifications into the library, we can also incorporate them directly into the application, avoiding the operating system altogether.

**Intercept System Calls:** Most modern operating systems provide the functionality of intercepting system calls at the user level. A process can be notified when another process enters or exits selected system calls. While the original motivation for this functionality was debugging and tracing of system calls, this mechanism can also be used

| Method                     | Examples and References   |
|----------------------------|---|
| Change Operating System    | Sprite [NWO88], Plan 9 [PPT90]  |
| Device Driver              | AFS, NFS, SLIC [GPA96] & WebFS [VDA96]  |
| Network Server             | ftp2nfs [Gsc94], Alex [Cat92]   |
| User-level Plug-Ins        | extended OS: SLIC [GPA96], UserFS [Fit96]<br>flexible/extendible OS: SPIN [BSP <sup>+</sup> 94], Exokernel [EKO95],<br>Newcastle Connection [BMR82], Prospero [NAU93], Condor [Con95] |
| Statically Linked Library  | Jade [RP93], IFS [EP93]   |
| Dynamically Linked Library | Ang-e-ftp [Nor]   |
| Application Specific       | Interposition Agents [Jon93], Confinement [GWTB96], Ufo   |
| Intercept System Calls     |   |

Table 1: Different methods of extending operating system functionality and examples.

| Method                     | Modify OS source | Requires root access | Re-compile applications | Re-link applications | Range of applications | Performance overhead |
|----------------------------|------------------|----------------------|-------------------------|----------------------|-----------------------|----------------------|
| Change Operating System    | X                | X                    |                         |                      | all                   | very low             |
| Device Driver              |                  | X                    |                         |                      | all                   | very low             |
| Network Server             |                  | X                    |                         |                      | all                   | medium               |
| User-level plug-ins        | once             |                      |                         |                      | all                   | low-high             |
| Statically Linked Library  |                  |                      | X                       | X                    | user                  | low                  |
| Dynamically Linked Library |                  |                      |                         | some                 | dyn. linked           | low                  |
| Application Specific       |                  |                      | X                       | X                    | single                | low                  |
| Intercept System Calls     |                  |                      |                         |                      | all (no setuid)       | high                 |

Table 2: Different methods of extending operating system functionality and their limitations.

to alter their behavior. This mechanism, which first was used in the context of Mach to implement interposition agents [Jon93], forms the basis for our Ufo implementation.

Table 1 lists examples of the above approaches, while Table 2 summarizes their limitations and identifies the context in which they can be applied. We wanted an approach which works with most existing applications without the need for recompiling, and more importantly, which can be used without requiring root access. Therefore we decided to use the mechanism of intercepting system calls.

## 2.2 Related OS Extensions

The project that is the closest to our own is the work on interposition agents [Jon93] which also makes use of the mechanism of intercepting system calls. Interposition agents provide a general system call tracing toolbox, which allows different system calls to be intercepted and handled in alternate ways, as we do in Ufo. Three example agent applications were implemented: spoofing the time of day, tracing system calls (as in truss), and transparently merging the contents of separate directories. The interposition agents work is based on Mach. While Mach is a Unix variant, it was designed to be more

flexible and extensible. In particular, when calls are intercepted in Mach 2.5, they can be redirected to the process' own address space. Thus, the interposition agents are run in the user process' own memory. Approaches that use a more standard Unix, such as ours, are more constrained (and more complicated to implement) since it is more difficult to access the user process state from outside of the process' address space.

Another research project that uses the Unix trace mechanism for implementing an OS extension is Janus [GWTB96] which provides a secure, confined environment for running untrusted applications safely by intercepting and selectively denying system calls. Like ours, the Janus implementation has been designed for Solaris.

A lot of current research deals with designing operating systems such that they allow for easier and more efficient user-level extension. Engler et al. [EKO95] carry the Mach micro-kernel methodology [ABB<sup>+</sup>86] further by removing as much kernel abstraction as possible from the OS. This pushes the kernel/user-level boundary as low as possible, placing most of the OS services outside of the kernel. Another approach, taken by VINO [SESS94] and SPIN [BSP<sup>+</sup>94], is to allow injection of user-written kernel extensions into the kernel domain. A discussion of the issues involved can be found in [SS96].

Another recent project, SLIC [GPA96], is an OS extension to Solaris that allows for plug-ins at both the user and the kernel level.

We now discuss operating system extensions specific to our particular application: remote file transfer.

## 2.3 OS Extensions for Remote File Systems

There are a number of systems that provide transparent access to remote resources on the Internet, many of which have been very successful. Examples include NFS [SGK<sup>+</sup>85], AFS [MSC<sup>+</sup>86], Coda [SKK<sup>+</sup>90], ftpFS in Plan 9 [PPTT90] and Linux [Fit96], Sprite [Wei91, NWO88], WebFS [VDA96], Alex [Cat92], Prospero [NAU93], and Jade [RP93]. They all have one significant drawback, however: they either require root access or modifications to the existing operating system, applications or libraries. Ufo is distinct in that it requires no such modifications to any existing code and runs entirely at the user-level.

There are a few systems for global file access that run entirely at the user-level and are user-installable. They are also similar to Ufo in that they extend a local file system to provide uniform and transparent access to heterogeneous remote file servers. Prospero [NAU93] and Jade [RP93] both provide access to NFS and AFS file systems, and to FTP servers. Prospero runs at user-level by replacing standard statically linked libraries. This avoids changes to the operating system, but requires re-linking of existing binaries. Jade [RP93] uses dynamic libraries instead and allows most dynamically linked binaries to run unmodified. Changing application libraries works well for most applications, especially when combined with dynamic linking. The drawback of this approach is that it does not work for statically linked applications not owned by the user as well as for applications that circumvent the standard libraries and execute system call instructions directly.

Other global file systems also run at the user level, but are not user-installable, since they require extensions to the operating system itself, which in turn requires root access. One such example is WebFS [VDA96], a global user-level file system based on the HTTP protocol. To run at the user level, WebFS relies on the OS extensions provided by SLIC [GPA96], which implements a call-back mechanism to a user process. (WebFS also requires the HTTP server be extended with a set of CGI scripts that service requests.) Similar to SLIC, UserFS [Fit96] is an OS extension that enables user-level file systems to be written for Linux. While installing UserFS itself requires kernel recompilation, installing new file modules, such as ftpFS, does not. Plan 9 [PPTT90] also includes an FTP based file system (also called ftpFS). At least two projects provide access to FTP servers by implementing an NFS server that functions as an FTP-to-NFS gateway.

Alex [Cat92] supports read-only access to anonymous FTP servers, while [Gsc94] additionally allows read and write access to authenticated FTP servers.

## 3 Catcher Implementation

In this section we discuss the details of our implementation of the Catcher inside Ufo. We start by describing the high-level architecture and the role of the Catcher in Ufo.

### 3.1 The Ufo Architecture

Ufo is a user-level process which provides file system services to other user-level processes by *attaching* to them. Once attached to a subject process, it intercepts system calls and services them if they operate on remote files. The application is unaware of the existence of the Ufo, but, with Ufo's help, it can operate on remote files as if they were local.

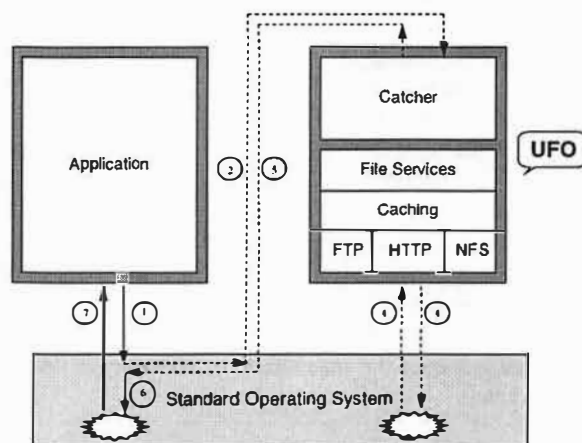


Figure 2: General architecture of Ufo.

Ufo is implemented in two modules: the Catcher and the Ufo module (Figure 2). The Catcher is responsible for intercepting system calls and forwarding them to the Ufo module. The Ufo module implements the remote file system and consists of three layers: the File Services layer which identifies remote files, the Caching layer, and the Protocol layer containing different plug-in modules implementing the actual file transfer protocols.

Figure 2 shows the steps involved in servicing a remote file request. When the application issues a system call (1), it can go directly to the kernel or, if it is file-related, get intercepted by the Catcher (2). For intercepted calls, Ufo determines whether the system call operates on a remote or a local file, possibly using kernel services (3,4). If the file is local, the request proceeds unmodified. If the file is remote, Ufo creates a

local cached copy, patches the system call by modifying its parameters, and lets the request proceed to the kernel (5). After the request is serviced in the kernel (6), the result is returned to the application (7). The return from the system call may also be intercepted and patched by Ufo, though the figure does not show this.

### 3.2 Catcher Implementation Details

In our Solaris implementation, the Catcher monitors user processes using the `/proc` virtual file system [FG91]. This is the same method used by monitoring programs, such as `truss` or `strace`, which are also available on a number of other UNIX platforms, including Digital Unix, IRIX, BSD or Linux. The System V `/proc` interface allows us to monitor and modify an individual process by operating on the file associated with a user process.

In particular the Catcher attaches to a subject process `pid` by opening the `/proc/pid` file. Once attached, the Catcher uses `ioctl` system calls on the open file descriptor to control the process. It can instruct the operating system to stop the subject process on a variety of events of interest. In Ufo there are two events of interest: system call entry into the kernel, and system call exit from the kernel. Once a subject process has stopped on an event of interest, the Catcher can read and write the registers and read and write in the address space of the process. The Catcher uses this to examine and modify the parameters or the result of system calls like `open`, `stat`, and `getdents`. Finally, the `/proc` interface allows us to restart the execution of a stopped process. Figure 3 summarizes how the discussed functionality is used in the Catcher.

```
connect to subject process
register the set of system calls to intercept
while subject process is running do
    wait for process to stop on system call entry or exit
    determine system call number and its parameters/result
    call handler function for this system call
    if system call is fork then
        begin monitoring new child process
    resume system call
endwhile
```

Figure 3: Outline of the Catcher algorithm.

Conceptually, Ufo implements the system calls intercepted by the Catcher, but in practice Ufo does not service them directly. Although implementing the system calls directly in Ufo would be possible, it would require reimplementing existing OS functionality. Instead, we

“patch” the system calls by (i) modifying the call's parameters, (ii) changing the file system state (e.g., fetching a file from a remote server) and (iii) modifying the result returned from the operating system. A good example for the first two actions is the `open` system call. On the entry of an open call, we may have to modify the file name string to point to the locally cached copy. Before allowing the system call to continue, the Catcher may have to wait for Ufo to download the file from the remote site. Implementing the name change is somewhat complicated since we must modify the user's address space. We cannot just change the filename in place since the new filename might be longer than the old one. Also, the filename could be in a segment which is read-only or shared among threads. Currently we solve these problems by writing the new file name in the unused portion of the application's stack and changing the system call argument to point to the new string.

The open system call needs to be intercepted on exit from the kernel as well. Although the returned result is not modified, Ufo must remember the correspondence between the returned file handle and the file name, which is needed when the file is closed.

Besides file related system calls, there are several others that must be intercepted. For example, to track child processes we intercept the `fork` system call. Given the child pid, we can open its associated `/proc` file and monitor it as well. System V allows the set of trapped system calls to be automatically inherited from parent to child, so this setup is only needed for the initial process.

### 3.3 User-Level Restrictions

Implementations of file system functionality at the user level must obey some restrictions since user-level processes cannot perform arbitrary actions and cannot access the whole file system related state of the kernel.

One problem is that the Catcher cannot control `setuid` process since the security policy of the operating system disallows user-level processes from attaching to other users' processes. In practice we have found this not to be a problem since very few programs are installed with `setuid`. And for most of those programs, e.g., `rlogin`, it is not clear whether one really needs the file system extensions. In the current implementation, whenever the Catcher detects that a subject process is about to spawn a `setuid` program, it just does not trace the child process.

In Solaris, the `/proc` interface allows the controlling process to write in the subject process' address space. This is important if the Catcher needs to change some system call arguments such as filename strings for Ufo. If we are to port the Catcher to other operating systems that do not provide the capability of writing into the subject process, we would not be able to implement this feature. Although writing in the user process is not neces-

sary for the basic functionality of Ufo, a Ufo implementation on such an operating system would have some limitations. Features such as the URL naming scheme and mountpoints in the root directory require changing of string arguments to system calls and therefore would not be possible to implement (see Subsection 4.1).

Another problem arises when the Catcher process is killed. Being a regular user-level process, the Catcher cannot protect itself against the SIGKILL signal. There is no graceful way to handle such a situation if the subject processes running under the Catcher continue working on remote files. In the current Ufo implementation the subject process will be trapped on the next intercepted system call and stay trapped until killed.

### 3.4 Catcher Discussion

The Catcher mechanism allows us to create a personalized operating system. Requests made to the kernel can be re-interpreted, in effect allowing individual users to run their own OS. Any user can use the “new” OS without having to modify the original operating system or needing root access. Although, the current Catcher only intercepts system calls, System V allows the user to also intercept and act on signals and hardware faults. This allows for a wide range of OS functionality to be extended using the Catcher mechanism. Other potential uses of the Catcher for personalized OS extensions include encrypting file systems, file systems which store files in compressed form, confined execution environments for running untrusted binaries [GWTB96], virtual memory paging [DWAP94, FMP<sup>+</sup>95], and process migration [Con95].

A potential concern with our approach is its performance overhead. Indeed, intercepting individual system calls is quite expensive and for some OS extensions this overhead would be unacceptable. Nevertheless, Ufo is an example that there are OS extensions for which the Catcher mechanism works well. Our performance analysis shows that Ufo introduces moderate overhead for common applications. This is due to the fact that typical applications issue relatively few system calls, and not all system calls are intercepted in Ufo.

## 4 Ufo's Global File System Module

Ufo provides read and write access to FTP servers and read-only access to HTTP servers. The remote file access functionality is implemented in Ufo's file system module which is responsible for resolving remote file names, transferring files, and caching.

### 4.1 Naming Strategies

Ufo supports three ways of specifying names of remote files: (i) through a URL, (ii) through a regular filename implicitly containing the remote host, user name, and access mode, and (iii) through mount points.

The first way to specify a remote file is through its URL syntax. Unfortunately, some applications cannot handle URL names. Make and gmake cannot handle the colon in the URL, while Emacs considers // to be the root of the file system and thus discards everything to the left.

To alleviate these problems we also support specifying a remote file through a regular filename. The general syntax is /protocol/user@host/filename where protocol is the file transfer protocol, e.g., ftp or http.

Lastly, Ufo allows the user to specify explicit mountpoints for remote servers or access protocols in a .uforc file. For example, the line

```
local /csftp remote /  
machine ftp.cs.ucsb.edu method FTP
```

specifies that accesses relative to /csftp refer to the root directory of the ftp.cs.ucsb.edu anonymous FTP server. The user can also specify mountpoints for access methods. In fact that is how the second naming scheme is implemented: if the user does not explicitly specify a mount point for the HTTP method, for example, Ufo uses the implicit mountpoint:

```
local /http method HTTP
```

Similarly to Sprite [NWO88], we have implemented mount points using a prefix table which, given a filename, searches for the longest matching prefix in the list of mount points.

Ufo also supports symbolic links. A user can create links to frequently accessed remote directories. While links simplify accesses to remote files, they actually present quite an implementation challenge, since they require following all link components to determine the true name of a file.

### 4.2 Accessing Remote Files and Directories

Ufo transfers only whole files to and from the remote file system. Whenever Ufo intercepts the *open* system call for a remote file, it ensures that a local copy of the file exists in the cache, and then redirects the system call to the local copy. *Read* and *write* system calls don't even have to be intercepted since they operate on file descriptors returned by the *open*; they will correctly access the local copy in the cache. Finally, on a *close* system call, Ufo checks whether the file has been modified and if so,

stores the file back to the server (the store may be delayed if write-back caching is in effect). Ufo uses whole file transfers for two reasons: this minimizes the number of system calls that need to be intercepted, and protocols such as FTP only support whole file transfers.

When an application requests information about a remote file, e.g., through a *stat* or *lstat* system call, Ufo satisfies the request by creating a local *file stub* and redirecting the system call to it. The file stub has the correct modification date and size of the remote file but contains no actual data.<sup>3</sup> With this approach Ufo neither has to re-implement the *stat* system call, nor download the whole file. Only if the application wants to open a file stub later, will Ufo actually download the remote file. Similarly, when a system call such as *getdents* (get directory entries) is issued on a remote directory, Ufo creates a copy of the directory in the local cache and puts file stubs in it. Then, it redirects the system call to the so created skeleton directory.

### 4.3 Caching and Cache Consistency

Since remote data transfers can be quite slow, Ufo implements caching of remote files to achieve reasonable performance. Instead of downloading a file each time the user opens it for reading, Ufo keeps local copies of previously accessed files. Ufo can reuse the local copy on a subsequent access, as long as it is up-to-date. Similarly, we use write-back caching which delays writing a modified file back to the remote server. While files are the primary objects cached, Ufo also caches directory information (directory contents), and file information (size, modification time, permissions). The FTP module additionally caches open control connections. Since establishing a new connection to the remote server for each transfer is expensive, we reuse open control connections by keeping them alive for a period of time after a transfer has completed.

The cache consistency policy governs whether we are allowed to use a local copy on a read, and whether we can delay the write-back of a modified file. To efficiently support a wide range of usage patterns, Ufo provides an adjustable consistency policy based on timeouts (a read and write delay). The policy guarantees that (i) when a file is opened it is no more than  $T_{read}$  seconds out of date; and (ii) changes made to a file will be written back to the server within  $T_{write}$  seconds after the file is closed. To verify that a local file is up to date (i.e., is not stale), Ufo checks whether the file on the remote site has changed (*validate on open*).  $T_{read}$  and  $T_{write}$  can have a zero value. In this case files opened for reading

<sup>3</sup>Creating a stub is done by seeking to the desired position in a newly created file and then writing a single byte. On most file systems, the so created stub occupies a small amount of disk space, independent of the reported file size.

are never stale and modified files are written back to the server immediately after they are closed.

The write timeout of a file is always a certain number of seconds. The read timeout can optionally be specified as a percentage of the file's age as in Alex [Cat92]. This method is based on the observation that older files are less likely to change than newer files. Therefore older files need to be validated less often. Files can have individual timeouts and Ufo provides mechanisms for the user to define default timeouts for all files, or for all files on a server. This allows the user to adjust the tradeoff between performance and consistency based on known usage patterns. For example, when mounting read-only binaries large read timeouts can be used since these files change rarely.

### 4.4 Authentication and Security

Ufo relies on the underlying access protocols for authentication. Currently, passwords are only required for authenticated FTP servers and are not needed for HTTP and anonymous FTP accesses. Ufo allows the passwords to be stored in the *.uforc* or *.netrc* files, or alternatively, Ufo asks for the password on the first access to a remote server.

Since Ufo is running entirely at the user level with the access permissions of its owner, it does not introduce new security problems in the system. The only potential security concern is to ensure that other users do not gain undesired access to the files in the private Ufo cache. We avoid this problem by creating the topmost cache directory with read and write permissions for the owner only.

### 4.5 Implementation Trade-Offs

In implementing Ufo, we tried to minimize the amount of operating system functionality that we had to reimplement. First, we attempted to minimize the number of intercepted system calls in order to minimize the execution overhead that Ufo introduces. This led to the whole file caching policy. Second, we wanted to minimize the implementation effort by modifying/reimplementing as few system calls as possible. This led to our decision to create file stubs and skeleton directories for the *stat* and *getdents* calls.

Of course, there is a trade-off between execution overhead and implementation effort. For example, the advantage of creating file stubs and skeleton directories is that we do not have to reimplement the *stat* and *getdents* system calls. The disadvantages are that creating file stubs may have high overhead. Also for efficiency, we rely on the support for holey files by the local file system. For example, on our machines the */tmp* file system does not support holey files, thus

if we use /tmp for the Ufo cache the stubs for large files do use all the disk space indicated by their size. The NFS-mounted file systems at our site do support holey files, but the stub creation there is an order of magnitude slower than on /tmp. For these reasons we are considering implementing the *stat* and *getdents* system calls completely inside the next version of Ufo to improve its performance. In fact, we are already partially implementing (patching) the *getdents* system call in order to support Ufo mountpoints in user-unwritable areas such as the root directory.

Transferring only whole files introduces three well known problems for extremely large files [Cat92]. First, when only a small fraction of a file is actually accessed, a lot of unnecessary data may be transferred. Second, the whole file has to fit on the local disk. In practice we don't expect these two problems to occur frequently. With the exception of databases, most applications tend to access files nearly in their entirety [BHK<sup>+</sup>91]. Furthermore, Ufo allows any local file system to be used for file transfers, thus reducing the danger of insufficient local disk space. A third problem comes from our decision not to intercept the *read* and *write* system calls. In our approach the open call blocks until the whole file has been transferred. It is possible to intercept and handle *read* and *write* system calls in Ufo. The benefit is that *open* would not always block:<sup>4</sup> reads that operate on the already present part of a file could be executed without waiting for the completion of the whole transfer (see Alex [Cat92]). The drawback is that intercepting *read* and *write* calls incur a high overhead and requires extra implementation effort.

## 5 Performance Measurements

The main goal of our performance analysis is to measure the overhead introduced by the Catcher mechanism in Ufo. This information is necessary to determine the usability of our method for operating system extension.

We first present the results of several microbenchmarks, which measure the overhead of intercepting individual Unix system calls. To demonstrate the overall impact of this overhead on whole applications we also present measurements for a set of file system benchmarks and a set of real-life applications. While the microbenchmarks show that intercepting system calls is expensive, the real-life applications exhibit much lower overhead.

All tests were run on a 143 MHz Sun Ultra 1 workstation with 64 megabytes of main memory running Solaris 2.5.1.

<sup>4</sup>Several other system calls like *lseek* also have to be intercepted for this to work.

### 5.1 Microbenchmarks

The microbenchmark results present the user-perceived run times (measured as wall clock times) for *open*, *close*, *stat*, *read*, *write*, and *getpid* system calls. The results are shown in Table 3. The columns show the numbers for the normal user program, for the Catcher-monitored program (Catcher only, no calls to Ufo functions), and for the Ufo program (Catcher and Ufo functionality). In the latter case, we examine the run times for a local file, for a cached remote file and for a remote file that has not been cached.

The *Catcher only* and *Ufo local file* numbers are of special significance. They show the cost of running a process under the Catcher or under Ufo when the process accesses local files only and does not require any of the extended OS functionality. This is the the fundamental overhead introduced by our method of extending the OS. The numbers for remote files are a measure of the combined effect of our remote file system implementation, our caching policy, the efficiency of the underlying access protocol (FTP in this case), and the quality of the network connection.

In order to measure the cost of the Solaris system calls themselves and not the network speed or the NFS overhead, we used the local /tmp file system. Accesses to /tmp are very fast and do not involve disk, network traffic or protocol overhead. As a result the microbenchmarks present the Catcher and Ufo overhead in the worst-case scenario. The relative Catcher and Ufo overhead for accessing non-cached NFS files, for example, is much lower.

The microbenchmarks were run on a lightly loaded workstation by taking the *wall-clock* time just before and just after the system call. The timing was done using the high resolution timer *gethrtime* which has a resolution of about 0.5 microseconds on the Ultra 1 workstation. Since individual system calls are very fast, normal system activity such as interrupts and context switches distorts some of the measurements. This produces a small percentage of outliers that are several times larger than the rest of the measurements. To ensure we do not include unrelated system activity in our measurements, in each test run we recorded 100 measurements and discarded the highest 10% of them. The remaining times were then averaged. The numbers in the table are the arithmetic mean of five such runs. The standard deviation for the five runs was below 2% for all tests, except for *getpid*, for which the standard deviation was at most 6%.

The *Catcher only* numbers show the cost of intercepting system calls. The results are obtained by running the benchmark program under the control of the Catcher alone. The Catcher simply intercepts the *open*, *close* and

| System Call | Standard OS       | Catcher only       | Ufo local file     | Ufo remote cached   | Ufo remote no cache |
|-------------|-------------------|--------------------|--------------------|---------------------|---------------------|
| open        | 28 $\mu s$ (1.00) | 211 $\mu s$ (7.54) | 611 $\mu s$ (21.8) | 2383 $\mu s$ (85.1) | 531 $ms$ (18964)    |
| close       | 12 $\mu s$ (1.00) | 108 $\mu s$ (9.00) | 300 $\mu s$ (25.0) | 1530 $\mu s$ (128)  | 452 $ms$ (37667)    |
| stat        | 33 $\mu s$ (1.00) | 133 $\mu s$ (4.03) | 472 $\mu s$ (14.3) | 1744 $\mu s$ (52.8) | 198 $ms$ (6000)     |
| getpid      | 3 $\mu s$ (1.00)  | 5 $\mu s$ (1.67)   | 5 $\mu s$ (1.67)   | 5 $\mu s$ (1.67)    | 5 $\mu s$ (1.67)    |
| write 1b    | 23 $\mu s$ (1.00) | 26 $\mu s$ (1.13)  | 26 $\mu s$ (1.13)  | 26 $\mu s$ (1.13)   | 26 $\mu s$ (1.13)   |
| read 1b     | 25 $\mu s$ (1.00) | 28 $\mu s$ (1.12)  | 28 $\mu s$ (1.12)  | 28 $\mu s$ (1.12)   | 28 $\mu s$ (1.12)   |
| write 8K    | 97 $\mu s$ (1.00) | 101 $\mu s$ (1.04) | 101 $\mu s$ (1.04) | 102 $\mu s$ (1.05)  | 101 $\mu s$ (1.04)  |
| read 8K     | 75 $\mu s$ (1.00) | 78 $\mu s$ (1.04)  | 78 $\mu s$ (1.04)  | 79 $\mu s$ (1.05)   | 79 $\mu s$ (1.05)   |

Table 3: Run times in microseconds for various system calls for accessing files in /tmp (the numbers are the arithmetic mean of 5 runs, each executing 100 iterations). The numbers in parentheses represent the ratio normalized to the standard Solaris OS.

*stat* system calls executed by the benchmark program, and lets them continue immediately without modifying them. The *read*, *write* and *getpid* system calls are not intercepted at all. Even though one may expect that these system calls will not be affected, they do incur a small overhead: whenever there is even a single intercepted system call for a process, the operating system takes a different execution path for all system calls of that process, independent of whether they are intercepted or not. The results demonstrate that for *read* and *write* of 1 byte blocks this overhead is small and for 8K blocks it is negligible. Because *getpid* is so fast, it has a substantial relative overhead, but still only 2  $\mu s$  total. On the other hand, system calls that must be trapped by the Catcher incur a factor of 4–9 overhead. During this extra time, control is passed from the program to the Catcher, (which performs *ioctl* calls to read information from the */proc* file system), and then back again.

The *Ufo local file* column shows how much extra overhead is introduced by Ufo in addition to the Catcher. The benchmark program is running under Ufo and is accessing local files only. Even though no remote files are accessed, Ufo still introduces some overhead in addition to the Catcher overhead. The extra overhead comes from the analysis of the parameters of the intercepted system calls. For system calls that reference a file, Ufo determines whether the file is indeed local or remote. Since a system call does not necessarily take an absolute path name as an argument, Ufo has the responsibility of determining it. Determining the true filename can involve a number of *stat* system calls, similar in flavor to the *pwd* command, and this can add a noticeable overhead.

The remaining two columns measure the overhead of Ufo when working with remote files. These numbers are measured as with *Ufo local file*, except that the accesses are to remote files. For the *Ufo remote cached* tests, a locally cached copy of the remote file is accessed. Note that in either case (cached or uncached), the *read* and

*write* system calls operate on the locally cached copy of the file. Thus, these numbers are consistent across all of the tests. On the other hand, *open* and *stat* calls to uncached remote files require remote accesses, and the overhead increases dramatically when Ufo uses the FTP protocol to retrieve the file. This overhead is almost entirely determined by the quality of the network connection and the FTP protocol. In our measurements we accessed files located at UC Berkeley. From a UC Santa Barbara machine, opening a remote file of size 1024 bytes residing at a UC Berkeley host requires 531 $ms$  using FTP. Closing the same remote file after modifying it takes 452 $ms$  since the file must be written back to the remote server. If the file is cached, the *open*, *close* and *stat* overhead is much smaller, but it still has roughly four times the overhead compared to a local file. This is due to two reasons: the additional work to manage the cache, and several remaining inefficiencies in our prototype implementation which will be corrected in future versions of Ufo.

## 5.2 File System Benchmarks

Table 4 reports the absolute execution times in seconds for two file system benchmarks run on the local /tmp file system with and without Ufo and on a remote FTP-mounted file system with and without caching. For these tests, the FTP host was a machine on the local 100Mbit/s Ethernet network. The remote tests with caching were with a warm cache and read and write delays set to infinity. Thus, these measurements represent the best-case scenario for remote files. For the remote tests without caching, the read and write delays were set to zero, forcing every *open*, *close* and *stat* system calls to go to the remote site. These tests are the worst-case scenario for accessing remote files under Ufo.

*lostone* and *Andrew* are standard file system benchmarks. We chose these as examples of applications that execute a lot of file system calls that Ufo intercepts

| Benchmark | System calls total | System calls intercepted | Standard OS             | Ufo local file | Ufo remote cached | Ufo remote no cache |
|-----------|--------------------|--------------------------|-------------------------|----------------|-------------------|---------------------|
| iostone   | 99203              | 48762                    | 3 s (1.00)              | 25 s (8.33)    | 102 (34)          | 7744 (2581)         |
| andrew    |                    |                          |                         |                |                   |                     |
| makedir   | 641                | 465                      | 0 s <sup>†</sup> (1.00) | 1 s (-)        | 1 s (-)           | 3 s (-)             |
| copy      | 9596               | 6999                     | 3 s (1.00)              | 6 s (2.00)     | 7 s (2.33)        | 68 s (22.7)         |
| scandir   | 8674               | 4115                     | 6 s (1.00)              | 7 s (1.17)     | 10 s (1.67)       | 61 s (10.2)         |
| read all  | 12907              | 6084                     | 10 s (1.00)             | 13 s (1.30)    | 16 s (1.60)       | 68 s (6.80)         |
| make      | 8407               | 3107                     | 12 s (1.00)             | 14 s (1.17)    | 16 s (1.33)       | 22 s (1.83)         |
| total     | 40225              | 20770                    | 30 s (1.00)             | 40 s (1.33)    | 50 s (1.67)       | 321 s (10.7)        |

Table 4: Run times for the Iostone and Andrew file system benchmark programs with and without Ufo. Times are in seconds, with the ratios normalized to the standard OS shown in parenthesis. (<sup>†</sup>The Andrew benchmark reports its timing results with a resolution of 1 second. The 0 seconds in the table indicate a measurement between 0 and 1 second.)

and handles. The *Iostone* benchmark [IOS87] performs thousands of file accesses (opening, reading, and writing). Because of the large amount of file opens and closes, Ufo runs about 8 times slower on the local file system. The *Andrew* benchmark [HKM<sup>+</sup>88] measures five stages in the generation of a software tree. The stages (i) create the directory tree, (ii) copy source code into the tree, (iii) scan all the files in the tree, (iv) read all of the files, and finally (v) compile the source code into a number of libraries. For this benchmark the Ufo overhead on local files is a factor of 1.33, much lower than the overhead for *Iostone*. For both *Andrew* and *Iostone*, the results for the uncached remote tests are orders of magnitude worse than for the local /tmp file system. This is not not surprising since the network latency and the FTP protocol overhead are quite large compared to the fast accesses in /tmp.

### 5.3 Application Programs

We also tested Ufo with a number of larger Unix applications: *latex*, *ghostscript*, a *make* of the Ufo executable, and the integer applications from the SPEC95 benchmark suite. The results are shown in Table 5. As with the file system benchmarks, each test was run without Ufo, under Ufo on local files only, and under Ufo on remote files with and without caching.

The first set of benchmarks are programs that we run frequently. The *latex* test measures the time to latex three times a 20 page paper consisting of 8 tex files and then produce a postscript from the dvi file. The *make* test compiles Ufo itself using g++. The *ghostscript* test displays a 20 page postscript document. The table shows that *latex* and *make* perform a relatively large number of system calls that Ufo intercepts, mainly *open*, *close*, and *stat*. This results in Ufo overheads of 24% and 22% respectively, when run locally, and higher overheads, when run remotely. The remote overheads, while large, should

be acceptable to the user, since accessing remote files is expected to cost extra time. The local overheads on the other hand, are incurred only because the application is running under Ufo even though it is not using any of its functionality. To avoid unnecessary local overhead, applications that only access local files can be run without Ufo, and Ufo can be detached from applications once they stop accessing remote files.

The *ghostscript* test on the other hand performs few calls that Ufo intercepts and never writes to the remote server; as a result the Ufo overhead is very low even in the remote test. This sort of overhead should be unnoticeable to the user.

The last eight tests are the integer applications from the SPEC95 benchmark suite. These were chosen as examples of compute intensive applications that do not perform extensive file system operations. For these applications the observed overhead is very small in the local and even in the remote tests. Small perceived overheads should also be expected for interactive applications such as text editors since the user is not likely to notice the difference between 28 $\mu$ s and 611 $\mu$ s when opening a local file.

### 5.4 Summary of Experimental Results

As expected, we find that intercepting system calls can be very expensive, and remote accesses are orders of magnitude higher than local accesses. For programs such as the *Iostone* benchmark — which performs many *open*, *close* and *stat* calls — the Ufo overhead for local files is too large to be ignored. Clearly, such programs should not be run under Ufo if they only access local files since this will incur a large overhead even though the program does not utilize any of the extended functionality. If remote files need to be accessed, then programs like *Iostone* will run slow, but this is mainly due to the network latency and access protocol overhead which

| Application  | System calls total | System calls intercepted | Standard OS   | Ufo local file | Ufo remote cached | Ufo remote no cache |
|--------------|--------------------|--------------------------|---------------|----------------|-------------------|---------------------|
| latex        | 7638               | 4396                     | 13.0 s (1.00) | 16.1 s (1.24)  | 17.6 (1.35)       | 90.0 s (6.92)       |
| make         | 22783              | 6762                     | 31.0 s (1.00) | 36.5 s (1.22)  | 38.5 (1.24)       | 104.8 s (3.38)      |
| ghostscript  | 5495               | 167                      | 4.1 s (1.00)  | 4.3 s (1.05)   | 4.4 (1.07)        | 5.6 s (1.37)        |
| 0.99.go      | 131                | 48                       | 833 s (1.00)  | 876 s (1.05)   | 880 (1.06)        | 914 (1.10)          |
| 124.m88ksim  | 161                | 18                       | 469 s (1.00)  | 470 s (1.00)   | 472 (1.01)        | 506 (1.08)          |
| 126.gcc      | 21234              | 1008                     | 343 s (1.00)  | 348 s (1.01)   | 348 (1.01)        | 365 (1.06)          |
| 129.compress | 40                 | 14                       | 344 s (1.00)  | 348 s (1.01)   | 348 (1.01)        | 344 (1.00)          |
| 130.li       | 250                | 60                       | 476 s (1.00)  | 478 s (1.00)   | 486 (1.02)        | 498 (1.05)          |
| 132.jpeg     | 1050               | 45                       | 484 s (1.00)  | 486 s (1.00)   | 482 (1.02)        | 498 (1.03)          |
| 134.perl     | 6019               | 42                       | 446 s (1.00)  | 454 s (1.02)   | 445 (1.00)        | 458 (1.03)          |
| 147.vortex   | 8490               | 27                       | 602 s (1.00)  | 604 s (1.00)   | 602 (1.00)        | 607 (1.01)          |

Table 5: Relative run times for some file system benchmarks and larger Unix applications. Times are in seconds, and the relative speed in parentheses. The first column shows the number of system calls executed by the application.

by far outweighs the Catcher and Ufo overhead as shown in Table 3. In this case Ufo proves to be a convenient tool. Furthermore, Ufo allows the user to dynamically attach to a running process and detach from it, so the choice of running under Ufo or not is always available. Applications that need access to remote files can have it, and the remaining processes will not incur any overhead.

Other applications, such as *make*, and *latex* incur a 22-24% overhead on local files — noticeable, but perhaps acceptable to the user even when the functionality of Ufo is not required. For remote files these applications incur overheads of 24% in the best case and 600% in the worst case depending on the kind of file caching used. In most cases the user expects that working on remote files would be slower, so the use of the extra functionality provided by Ufo should be worth the additional overhead, especially when the only alternative is to manually transfer files using FTP. Many other applications, such as compute intensive programs or text editors, make infrequent use of the system calls trapped by Ufo (though they may use other calls such as *read* and *write*). For such applications, user-perceived delays are much smaller: on the order of a few percent. In this case, running applications under Ufo makes no appreciable difference.

From these observations we can draw the conclusion that the Catcher is a good tool for implementing operating system extensions that require the interception only of relatively infrequent system calls. An example of such an extension is the Ufo file system when running real-life applications. On the other hand, this method is not ideal for extensions which intercept frequently occurring system calls.

Finally, we would like to mention that we are aware of some opportunities for improving the current Catcher prototype and many opportunities for improving Ufo. For example, optimizing the filename check in Ufo (to

determine whether a file is local or remote) alone will result in a significant reduction of the running time.

## 6 Conclusions and Future Work

In this paper, we presented a general way of extending operating systems functionality, using the debugging and tracing facilities provided by many Unix operating systems. Selected system calls are intercepted at the user-level and augmented to obtain the desired functionality. This mechanism forms the basis for Ufo, a file system providing transparent access to remote files on FTP and HTTP servers. Ufo proved to be a useful tool which we now use daily. As our experimental results show, its overhead, while quite large for intercepted system calls, is acceptable for most applications.

We believe that our approach is a promising way for individual users to develop and experiment with future operating system extensions, since this can be done completely at the user-level. Essentially, each user sees a personalized version of the operating system, extensions do not affect other users and are compatible with existing applications as those need not be re-compiled or re-linked. In the past, operating systems research had a hard time to carry over to the general public. With our approach, researchers can make their extensions easily available, and users can run them without relying on the system administrator for installation.

There are plenty of avenues for future work and research. For example, we have several ideas on how to improve the performance of the Catcher and Ufo. We also plan to implement new protocol modules in Ufo, e.g. based on NFS, WebNFS, and the rlogin protocols. We have experimented with several other OS extensions suitable for cluster of workstation environments. For example, we have developed a prototype that attaches to a

process, checkpoints it, and then can restart it at a later time or migrate it to another processor. Similarly, we have a prototype Catcher which intercepts all forks and execs and sometimes decides to execute some processes on other workstations. While both tools are still at a very crude stage, we have already seen some of their potential benefits. Similar benefits can be expected for paging virtual memory to the memory of idle processors instead of to a slow local disk.

Another interesting research area is protected computing. The system calls define the capabilities a process has and resources it can obtain (memory, disk access, CPU time). We can use the Catcher to limit the resources a process can access or obtain. This approach, implemented in Janus [GWTB96], is especially interesting in the current development of global computing, where one user may run an untrusted binary fetched from the Internet.

Finally, we intend to generalize our design of the Catcher since it can not only intercept system calls, but also signals and hardware traps which are delivered to the application. We intend to build a Catcher toolbox which can be used for OS courses and research projects.

## 7 Acknowledgements

We would like to thank Urs Hölzle for insightful discussion of this paper and for helping test early versions of Ufo. We would also like to thank Dave Probert, Chad Yoshikawa, Roger Faulkner, Arvind Krishnamurthy, and the anonymous referees for their valuable feedback.

## References

- [ABB<sup>+</sup>86] M. Acetta, R. Baron, W. Bolowsky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for Unix development. In *Proceedings of the USENIX Summer '86 Conference*, July 1986.
- [BHK<sup>+</sup>91] M. G. Baker, J. H. Hartmann, M.D. Kupfer, K. W. Shirrif, and J. K. Ousterhout. Measurement of a distributed file system. In *Proceedings of the 13th Symposium on Operating System Principles*, Pacific Grove, CA, 1991.
- [BMR82] D. R. Brownbridge, L. F. Marshall, and B. Randell. The Newcastle Connection, or UNIXes of the world unite! *Software — Practice and Experience*, 12, 1982.
- [BSP<sup>+</sup>94] B. N. Bershad, S. Savage, P. Pardyak, E. F. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th Symposium on Operating System Principles*, 1994.
- [Cat92] V. Cate. Alex — a global filesystem. In *Proceedings of the 1992 USENIX File System Workshop*, Ann Arbor, MI, May 1992.
- [Con95] The Condor Team. Checkpoint & migration of UNIX processes in the Condor distributed processing system. *Dr. Dobbs Journal*, February 1995.
- [DWAP94] M. Dahlin, R. Wang, T. Anderson, and D. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proceedings of the USENIX Conference on Operating System Design and Implementation*, May 1994.
- [EKO95] D. Engler, F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, December 1995.
- [EP93] P. R. Eggert and D. S. Parker. File systems in user space. In *Proceedings of the Usenix Winter 1993 Technical Conference*, Berkeley, CA, 1993. Usenix Association.
- [FG91] R. Faulkner and R. Gomes. The process file system and process model in UNIX system V. In *Proceedings of the 1991 USENIX Winter Conference*, 1991.
- [Fit96] J. Fitzhardinge. Userfs: A user file system for Linux. <ftp://sunsite.unc.edu/pub/Linux/ALPHA/userfs/userfs-0.9.tar.gz>, 1996.
- [FMP<sup>+</sup>95] M. J. Feeley, W. E. Morgan, F. H. Pighin, A. R. Karlin, and H. M. Levy. Implementing Global Memory Management in a Workstation Cluster. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.
- [GPA96] D. P. Ghormley, D. Petrou, and T. E. Anderson. SLIC: Secure loadable interposition code. Technical Report CSD-96-920, University of California, Berkeley, 1996.
- [Gsc94] M. Gschwind. FTP — access as a user-defined file system. *ACM Operating Systems Review*, 1994.
- [GWTB96] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A Secure Environment for Untrusted Helper Applications — Confining the Wily Hacker. In *Proceedings of the 1996 USENIX Security Symposium*, 1996.
- [HKM<sup>+</sup>88] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1), February 1988.
- [IOS87] IOSStone. A synthetic file system performance benchmark. Technical Report TR-074-87, Princeton University, 1987.
- [Jon93] M. B. Jones. Interposition agents: Transparently interposing user code at the system interface. In *Proceedings of the 14th Symposium on Operating Systems Principles*, New York, NY, 1993.

- [MSC<sup>+</sup>86] J. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. Rosenthal, and F. D. Smith. Andrew: A distributed personal computing environment. *Communications of the ACM*, 29(3), 1986.
- [NAU93] B. C. Neumann, S. S. Augart, and S. Upasani. Using Prospero to support integrated location-independent computing. In *Proceedings of the Symposium on Mobile and Location-Independent Computing*, Cambridge, MA, 1993.
- [Nor] A. Norman. *Ange-Ftp Manual*. Free Software Foundation, Inc.
- [NWO88] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1), February 1988.
- [PPT90] R. Pike, D. Presotto, K. Thompson, and H. Trickey. Plan 9 from Bell labs. In *Proceedings of the UKUUG Conference*, July 1990.
- [RP93] H. C. Rao and L. L. Peterson. Accessing files in an internet: The JADE file system. *IEEE Transactions on Software Engineering*, 19(6), June 1993.
- [SESS94] M. Seltzer, Y. Endo, C. Small, and K. Smith. An introduction to the VINO architecture. Technical Report TR34-94, Harvard University, 1994.
- [SGK<sup>+</sup>85] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network file system. In *Proceedings of the Summer USENIX conference*, June 1985.
- [SKK<sup>+</sup>90] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A highly available file system for a distributed workstation. *IEEE Transactions on Computers*, 39(4), 1990.
- [SS96] M. Seltzer and C. Small. A comparison of OS extension technologies. In *Proceedings of the 1996 Usenix Technical Conference*, San Diego, CA, January 1996.
- [TLC85] M. Theimer, K. Landtz, and D. Cheriton. Preemptable remote execution facilities for the V system. In *Proceedings of the 10th ACM Symposium on Operating System Principles*, December 1985.
- [VDA96] A. Vahdat, M. Dahlin, and T. Anderson. Turning the web into a computer. Technical report, University of California, Berkeley, 1996.
- [vEBBV95] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-net: A user-level network interface for parallel and distributed computing. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, December 1995.
- [VGA94] A. M. Vahdat, D. P. Ghormley, and T. E. Anderson. Efficient, portable, and robust extension of operating system functionality. Technical Report CS-94-842, University of California, Berkeley, December 1994.
- [Wel91] B. B. Welch. Measured performance of caching in the Sprite network file system. *Computer Systems*, 3(4), 1991.

# Network-aware Mobile Programs\*

M.Ranganathan, Anurag Acharya, Shamik D. Sharma and Joel Saltz  
Department of Computer Science  
University of Maryland  
College Park, MD 20740

## Abstract

*In this paper, we investigate network-aware mobile programs, programs that can use mobility as a tool to adapt to variations in network characteristics. We present infrastructural support for mobility and network monitoring and show how adaptalk, a Java-based mobile Internet chat application, can take advantage of this support to dynamically place the chat server so as to minimize response time. Our conclusion was that on-line network monitoring and adaptive placement of shared data-structures can significantly improve performance of distributed applications on the Internet.*

## 1 Introduction

Mobile programs can move an active thread of control from one site to another during execution. This flexibility has many potential advantages. For example, a program that searches distributed data repositories can improve its performance by migrating to the repositories and performing the search on-site instead of fetching all the data to its current location. Similarly, an Internet video-conferencing application can minimize overall response time by positioning its server based on the location of its users. Applications running on mobile platforms can react to a drop in network bandwidth by moving network-intensive computations to a proxy host on the static network. The primary advantage of mobility in these scenarios is that it can be used as a tool to adapt to variations in the operating environment. Applications can use online information about their operating environment and knowledge of their own resource requirements to make judicious decisions about placement of computation and data.

For different applications, different resource constraints are likely to govern the decision to migrate, e.g. network latency, network bandwidth, memory availability,

server availability. In this paper, we investigate *network-aware* mobile programs, i.e. programs that position themselves based on their knowledge of network characteristics. Whether the potential performance benefits of network-aware mobility are realized in practice depend on answers to three questions. First, how should programs be structured to utilize mobility to adapt to variations in network characteristics? In particular, what policies are suitable for making mobility decisions? Second, is the variation in network characteristics such that adapting to them can be profitable? Finally, can adequate network information be provided to mobile applications at an acceptable cost?

In order to adapt to network variations, mobile programs must be able to decide when to move, what to move and where to move. There are three types of network variations which may be cause for migration: (1) *population* variations, which represent changes in the distribution of users on the network, as sites join or leave an ongoing distributed computation; (2) *spatial* variations, i.e. stable differences between in the quality of different links, which are primarily due to the hosts' connectivity to the Internet; and (3) *temporal* variations, i.e. changes in the quality of a link over a period of time, caused presumably by changes in cross-traffic patterns and end-point load. Spatial variations can be handled by a *one-time placement* based on the information available at the beginning of a run. Adapting to temporal and population variations requires *dynamic placement* which needs a periodic cost-benefit analysis of current and alternative placements of computation and data. Dynamic placement decisions have two partially conflicting goals: maximize the performance improvement from mobility and minimize the cost of mobility. If an opportunity for improving performance presents itself, it should be capitalized upon; however, reacting too rapidly to changes in the network characteristics can lead to performance degradation as the performance gain may not offset the mobility cost.

We investigate these issues in the context of *Sumatra*, an extension of the *Java*<sup>1</sup> programming environment [10]

\*This research was supported by ARPA under contract #F19628-94-C-0057, Syracuse subcontract #353-1427

<sup>1</sup>Java is a registered trademark of Sun Microsystems.

that provides a flexible substrate for adaptive mobile programs. Since mobile programs are scarce, we developed a mobile chat server for our experiments. This application, called *adaptalk*, monitors the latencies between all participants and locates the chat server so as to minimize the maximum response time. We selected this application since it is highly interactive and requires fine-grain communication. If such an application is able to take advantage of information about network characteristics, we expect that many other distributed applications over the Internet would be similarly successful. The resource that governs the migration decisions of *adaptalk* is network latency. To provide latency information, we have developed *Komodo*, a distributed network latency monitor.

To evaluate if mobile applications can take advantage of network-awareness, we examined the performance of *adaptalk* with and without mobility. Our evaluation had two main goals: (1) to determine the performance benefits, if any, of network-aware placement of the central chat server over a network-oblivious placement; and (2) to determine if dynamic placement based on online network monitoring provides significant performance gains over a one-time placement based on initial information. Our results are encouraging - they indicate that on-line monitoring and dynamic placement can significantly improve performance of distributed applications on the Internet.

The paper is organized as follows. Section 2 describes Sumatra and the programming model that it provides. Section 3 describes the design and implementation of *Komodo*. Section 4 describes the *adaptalk* application and the policy it uses to make mobility decisions. Section 5 describes our experiments and presents the results. Section 6 discusses the results and their implications. Section 7 describes related work and Section 8 provides our conclusions and plans for future work.

## 2 Sumatra: a Java that walks

Sumatra is an extension of the Java programming environment that supports adaptive mobile programs. Platform-independence was the primary rationale for choosing Java as the base for our effort. In the design of Sumatra, we have not altered the Java language. Sumatra can run all legal Java programs without modification. All added functionality was provided by extending the Java class library and by modifying the Java interpreter without affecting the virtual machine interface.

Our design philosophy for Sumatra was to provide the mechanisms to build adaptive mobile programs. Policy decisions concerning when, where and what to move are left to the application. The main feature that distinguishes Sumatra from previous systems [3, 11, 13, 23] that support

mobile programs is that *all* communication and migration happens under application control. Furthermore, combination of distributed objects and thread migration allows applications the flexibility to dynamically choose between moving data or moving computation. The high degree of application control allows us to easily explore different policy alternatives for resource monitoring and for adapting to variations in resources. We believe that the space of design choices for adaptive mobile programs is yet to be mapped out and such flexibility is important to help explore this space.

Sumatra adds two programming abstractions to Java: *object-groups* and *execution engines*. An object-group is a dynamically created group of objects. Objects can be added to or removed from object-groups. All objects within an object-group are treated as a unit for mobility-related operations. This allows the programmer to customize the granularity of movement and to amortize the cost of moving and tracking individual objects. This is particularly important in languages like Java because every data structure is an object and moving the state one object at a time can be prohibitively expensive. An execution-engine is the abstraction of a location in a distributed environment. In concrete terms, it corresponds to an interpreter executing on a host. Sumatra allows object-groups to be moved between execution-engines. An execution-engine may also host active threads of control. Currently, multiple threads on the same engine are scheduled in a *run-to-completion* manner. We plan to implement other scheduling strategies in future. Threads can move between engines.

The principal new operations provided by Sumatra are:

**Object-group migration:** Object-groups can be moved between engines on application request. As mentioned earlier, all objects within an object-group are treated as a unit for mobility-related operations. Objects in an object-group are automatically marshalled using type-information stored in their class templates. When an object-group is moved, all local references to objects in the group (stack references and references from other objects) are converted into *proxy references* which record the new location of the object. Some objects, such as I/O objects, are tightly bound to local resources and cannot be moved. References to such objects are reset and must be reinitialized at the new site. The class template for an object (and the associated bytecode) can be downloaded into an execution-engine on application request.

**Remote method invocation:** Method invocations on proxy objects are translated into calls at the remote site. Type information stored in class-templates is used to achieve RPC functionality without a stub compiler. Exceptions generated at the called site are forwarded to the

caller. Sumatra does not automatically track mobile objects. Requesting a remote method invocation on an object that is no longer at the called site results in an *object-moved* exception at the calling site. To facilitate application-level tracking, the exception carries with it a forwarding address. The caller can handle the exception as it deems fit (e.g., re-issue the request to the new location, migrate to the new location, raise a further exception and so on). This mechanism allows applications to locate mobile objects lazily, paying the cost of tracking only if they need to. It also allows applications to abort tracking if need be and pursue an alternative course of action.

**Thread migration:** Sumatra allows explicit thread migration using a `engine.go()` function that bundles up the stack and the program counter and moves the thread to the specified execution-engine. Execution is resumed at the first instruction after the call to `go`. To automatically marshal the stack, the Sumatra interpreter maintains a type stack parallel to the value stack, which keeps track of the types of all values on the stack. When a thread migrates, Sumatra transports with it all local objects that are referenced by the stack but do not belong to any object-group. Objects that belong to an object-group move only when that object-group is moved. Stack references to the objects that are left behind (i.e. were part of some object-group) are converted to proxy references. After the thread is moved to the target site, it is possible that its stack contains proxy references that point to objects that used to be remote but are now local. These references are converted back to local references before the call to `go` returns.

**Remote execution:** A new thread of control can be created by *reexec'ing* the `main` method of a class existing on a remote engine. The arguments for new thread are copied and moved to the remote site. Unlike remote method invocation, remote execution is non-blocking; the calling thread resumes immediately after the `main` method call is sent to the remote engine. Remote execution is different from thread migration as it creates a new thread at the remote site that runs concurrently with the original thread; thread migration moves the current thread to the remote site without creating a new thread. Concurrent threads communicate using calls to shared objects. The thread initiating a remote execution can share objects with the new thread by passing it references to these objects as arguments to `main`.

**Resource monitoring:** Sumatra provides a resource-monitoring interface which can be used by applications to register monitoring requests and to determine current values of specific resources. This interface is similar to an object-oriented version of the Unix `ioctl()` interface. When an application makes a monitoring request, Sumatra forwards the request to the local resource monitor. If

the monitor does not support the requested operation, an exception is delivered to the application.

**Signal handlers:** Sumatra allows applications to register handlers for a subset of Unix signals. Signals can be used by the external environment (the operating system or some other administrative process) to inform the application about urgent asynchronous events, in particular resource revocation. Using a handler, the application can take appropriate action including moving away from the current execution site.

## 2.1 Example

In this section, we provide a feel for the Sumatra programming model using a simple example. The task is to scan through a database of X-ray images stored at a remote site for images that show lung cancer. This task can be performed in two steps. In the first step, a computationally cheap pruning algorithm is used to quickly identify lungs that might have cancer. A compute-intensive cancer-detection algorithm is then used to identify images that actually show cancer.

One way to write a program for this task would be to download all lung images from the image server and do all the processing locally. If the absence of cancer in most lung images can be cheaply established, this scheme wastes network resources as it moves all lung images to the destination site. Another approach would be to send the selection procedure to the site of the image database and to send only the "interesting" images back to the main program. If the selection procedure is able to filter out most of the images, this approach would significantly reduce network requirements. A third, and even more flexible, approach would allow the shipped selection procedure to extract all the interesting images from the database but return only the *size* of the extracted images to the main program. If the size is too big, the program may choose to move itself to the database site and perform the cancer-detection computation there rather than downloading all the data. This avoids downloading most images at the cost of (possibly) slower processing at the server. On the other hand if the size of the images is small, the data can be shipped over and processed locally. Figure 1 shows code for the third approach. This program makes its decision to migrate in a rudimentary fashion; a more realistic version of this application would also take network bandwidth and the processing power available on both machines into consideration.

Sumatra assumes that a local resource monitor is available which can be queried for information about the environment. In the next section, we describe one such monitor which allows Sumatra applications to request information about network latency between any pair of sites that

```

****
filter_object = new Lung_filter();
cancer_object = new Lung_checker(filter_object);
myengine = System.rpc.myEngine();

// Create a engine at the xray database site.
remote_engine = new Engine("xrays.gov");
// Send the lung_filter class to the remote engine
remote_engine.downloadClass("Lung_filter");
// Create a new object group.
objgroup = new ObjGroup("lung_filter.group");
// Add the lung_filter_object to the object group
objgroup.checkIn(filter_object);
// Move the object group to the database site
objgroup.moveTo(remote_engine);

// a remote method call selects interesting xrays
size = filter_object.query(db, "DarkLungs");

// Are there too many images to bring over?
if ( size > too_many_images ) {
    // Migrate thread, process images and return.
    remote_engine.go();
    result = cancer_object.detect_cancer();
    myengine.go();
}
else {
    // there are only a few interesting xrays. Fetch them
    // and process locally.
    objgroup.moveTo(myengine);
    result = cancer_object.detect_cancer();
}

// display result locally
System.display(result);

```

Figure 1: Excerpt of a Sumatra program that adaptively migrates to reduce its network bandwidth requirements

run the monitor.

### 3 Komodo: a distributed network latency monitor

Komodo<sup>2</sup> is a distributed network latency monitor. The design principles of Komodo are: low-cost active monitoring and fault-tolerance. Active monitoring uses separate messages for monitoring; passive monitoring generates no new messages and piggybacks monitoring information on existing messages. An active monitoring approach is needed for adaptalk (described in the next section) as passive monitoring cannot provide information about links that are not used in the current placement but could be used in alternative placements. It is our working hypothesis that effective mobility decisions can be based on medium-term (30sec-few minutes) and long-term (hours) variations. At these resolutions, we believe that active monitoring can be achieved at an acceptable cost. This section briefly describes the design and implementation of Komodo. Further details about Komodo are presented in [18].

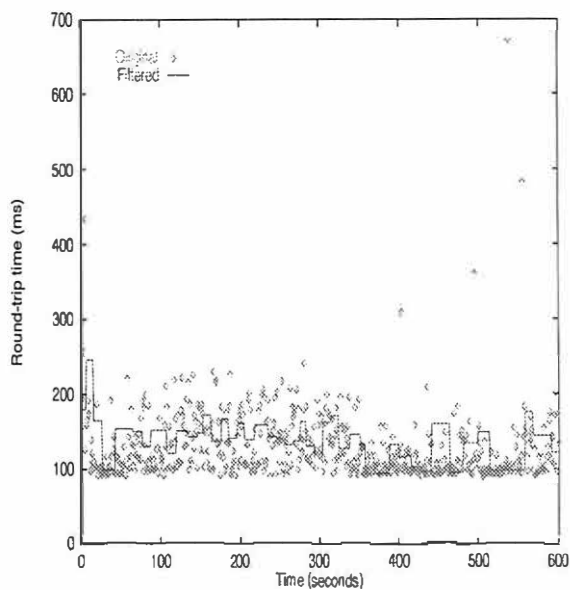
Komodo allows applications to initiate monitoring of network latency between any pair of hosts running the monitor; the application need not be resident on either of the hosts. Komodo is implemented as a user-level daemon that runs on every host participating in the computation. Applications pass monitoring requests to their local Komodo daemon. If the requested link includes the current host, the

local daemon handles the request. Otherwise, it forwards the request to the daemon on the appropriate host. Daemons determine network latency by sending 32-byte UDP packets to each other. If an echo is not received within an expected interval, (the maximum of the ping period or five times the current round trip time estimate) the packet is retransmitted. Using UDP for communication may, occasionally, lead to loss of messages. Message loss can lead only to a short-term loss of efficiency. As we expect monitoring requirements to be coarse-grained, the effect of packet loss should be small. Note that message loss is also a sign of network congestion and as such may be useful information for applications.

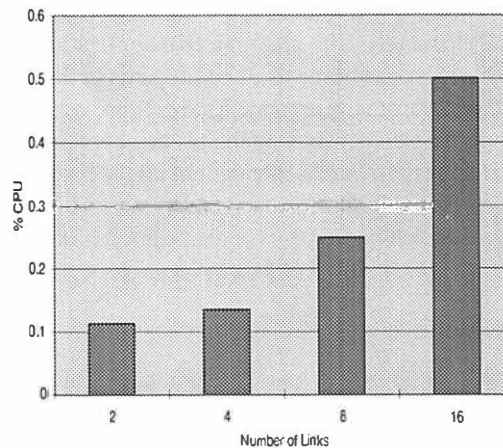
Applications that initiate a monitoring request can specify the frequency with which Komodo *pings* a link. Komodo enforces an upper bound on this frequency to keep the monitoring cost at an acceptable level. Applications need to refresh requests periodically to keep them alive; Komodo deactivates requests that have not been refreshed for longer than its *request-timeout* period.

Latency measures acquired by Komodo are passed through a filter before being provided to applications. This filter eliminates singleton impulses as well as noise within a jitter threshold (we use a jitter threshold of 10 ms, which is the resolution of most Unix timers). If the measure changes rapidly, a moving window average is generated. This filter was designed on the basis of our study of a large number of Internet latency traces (see Section 5.1) which revealed that: (1) there is a lot of short-term jitter in the latency measures but in most cases, the jitter is small; (2) there are occasional sharp jumps in latency that appear only for short time intervals; (3) occasionally, the latency measure

<sup>2</sup>Komododragons are a species of *monitor* lizards found on the island of Komodo which is close to both Java and Sumatra.



(a) Operation of the Komodo filter



(b) CPU utilization of Komodo

Figure 2: (a) The input to the filter is a 10-minute trace of one-per-second latency measures between `baekdoo.cs.umd.edu` and `lanl.gov`. Note that the four single-ping impulses towards the right end have been eliminated. (b) The CPU utilization is computed by dividing the (user+system) time by the total running time. Each experiment was run for 1000 seconds with one ping per second for all links.

fluctuates rapidly; (4) for time windows of 10 seconds or larger, the mode value (with a 10 ms jitter threshold) dominates. To elaborate the last point, in most time windows, 70-90% of the latency values fall within a jitter threshold of the most common value. Our filter attempts to find the mode for a recent time window. If there is no stable mode (as happens occasionally), it returns the mean. Figure 2(a) illustrates the operation of the filter.

Each daemon maintains a cache of current latency estimates for all links it is currently monitoring. This cache is maintained in a well-known shared memory segment and can be efficiently read by all Sumatra applications executing on the same machine. Cooperating Komodo daemons forward latency information in response to persistent remote requests. A latency estimate for a request received from another host is forwarded only when a new filtered estimate (different from the previous filtered estimate) is generated and is piggybacked onto a ping reply if possible. Currently, Komodo is implemented in C.

To address concerns about the cost of active monitoring, we measured the CPU utilization of Komodo for varying number of links. Results in Figure 2 (b) show that the maximum CPU utilization for up to sixteen links is about 0.5 %. The amount of data transferred is 512 bytes/second. This experiment was conducted on Sparc 5 machines (110MHz, 32 MB of memory) running SunOS Release 5.5.

#### 4 Adaptalk: An adaptive internet chat application

Adaptalk is a relatively simple network chat application built using Sumatra and Komodo. It allows multiple users to have an online conversation; new participants can join an ongoing conversation at any point; multiple independent conversations can be held. To ensure that all participants see the same conversation and that new participants can join ongoing conversations, a central server is used to serialize and broadcast the contributions.

Adaptalk is divided into three modules: handling keyboard events, managing the chat screen and coordinating the communication between participants. Each component is implemented by a separate object-group. Each host participating in the conversation runs two execution-engines, one houses the screen object-group and the other houses the keyboard object-group. The central server is implemented as a separate shared object-group, the `msgboard`, which can be placed on any host participating in the conversation. Each message issued by a participant starts from a keyboard object which invokes a remote method on the `msgboard`. The `msgboard` serializes incoming messages and issues a series of remote-execution requests, one per participant, which update the screen objects on all participants. In this case, remote execution is preferred to remote method invocation as there is no use-

ful return value and remote execution allows fast one-way communication.

Individual messages in *adaptalk*, and most other chat applications, consist of single lines of characters, usually no more than 50-60 characters. The goal of a chat application is to provide a short response-time to all participants so that a conversation can make quick progress. The response-time for a particular participant depends on the latency between it and the central server. Given the latencies of all the links, the primary knob that *adaptalk* can turn to maintain a low response-time for all participants is the position of the central server.

#### 4.1 Mobility policy

There are two main features of the *adaptalk* mobility policy. (1) continuous tracking of the instantaneously *most-suitable-site* and (2) deferral of server-motion till the potential for a significant and *stable* performance advantage has been seen. The first feature allows it to quickly take advantage of opportunities for optimization; the second helps ensure the gain is greater than the cost. The goal of *adaptalk* is to minimize the maximum response-time seen by any participant. The suitability of a participating machine as the location of the *msgboard* is characterized by the maximum network latency between it and all other participants. The machine that achieves the lowest measure is designated the *most-suitable-site*.

*Adaptalk*'s migration policy is shown in pseudo-code in Figure 3. This algorithm is run at the location that hosts the *msgboard* and recomputes the *most-suitable-site* each time a new message is posted by any participant. The *msgboard* maintains an array of counters, one for each potential location, which keep track of the number of times each location is found to be the *most-suitable-site*. The *msgboard* moves whenever: (1) the current site receives a very low score ( $< \text{loss\_threshold}$ ) over a given period (the *decision\_cycle*); or (2) a different site receives more than a threshold score (the *win\_threshold*). The first condition is used to quickly move away from locations that provide poor performance; the second condition is used to move the *msgboard* to locations that consistently promise better performance. The counters are reset whenever the *msgboard* moves, the *decision\_cycle* completes or a participant enters or leaves the conversation.

We expect three types of variations in the network characteristics which may be cause for migration: (1) *population* variations, which represent changes in the distribution of users on the network, as participants join or leave an ongoing conversation; (2) *spatial* variations, i.e. stable differences between latencies of different links; and (3) *temporal* variations, i.e. changes in the latency of a link over a period of time. *Adaptalk*'s migration policy can

adapt to all three types of variations. Consider the case with a fixed number of participants with significant spatial variation in network latency and little temporal variation. In this case, the migration algorithm rapidly recognizes the best location for the *msgboard*, but waits until this choice has been ratified over some period of time ( $\text{count}[\text{newloc}] > \text{win\_threshold}$ ) before moving it. As shown in Section 5, this policy allows *adaptalk* to effectively insure itself against poor *initial placement*. Once a good location has been found, the *msgboard* does not move, unless temporal variations or changes in population distribution cause another node to become a substantially better location (i.e.  $\text{count}[\text{newloc}] > \text{win\_threshold}$ ) or the current host to become a substantially bad choice (i.e.  $\text{count}[\text{curr\_engine}] < \text{loss\_threshold} \ \&\& \ \text{rounds} \% \text{decision\_cycle} == 0$ ). In such cases, the *msgboard* will move during the conversation. After initial experiments with *adaptalk*, we set the *win\_threshold* to be  $25 \times n$ , the *loss\_threshold* to be  $12 \times n$  and the *decision\_cycle* to be  $50 \times n$ . Here,  $n$  is the number of participants. The length of the *decision\_cycle* was set large enough to amortize the cost of movement in cases where large temporal variations or fluctuations in population distribution cause frequent repositioning.

```
.....
Get the all to all latency map from Komodo;
Find the site s that would minimize the max
    latency for messages posted to msgboard;
count[s] = count[s] + 1; rounds++;
let w be the site with the largest count;
let curr_engine be the engine which
    currently houses msgboard;
// Found a clear cut winner.
if (count[w] > win_threshold) return w;
else if (rounds % decision_cycle == 0) {
// Is the current engine an ok location ?
if (count[curr_engine] > loss_threshold) {
    clear count for each host;
    return curr_engine;
} else {
    // Current engine is a bad location.
    set new_host to the host with the
        maximum count;
    clear count for each host;
    return new_host;
}
} else return null; // cycle not yet over.
```

Figure 3: Decision Algorithm for *msgboard* placement used in *Adaptalk*. This algorithm is run at the location where the *msgboard* resides each time a message is posted.

## 5 Evaluation

To evaluate the performance impact of network-aware adaptation on the Internet, we performed two sets of experiments. First, we monitored round-trip times for 32-byte ICMP packets between a large set of host-pairs over several days. The goal of these experiments was to study the spatial and temporal variation in network latency on the Internet. Results from this study are presented in section 5.1.

Second, we measured the performance of three versions of *adaptalk* over long-haul networks, using traces collected during the Internet study. Our evaluation had two main goals: (1) to determine if network-aware placement of components of an application distributed over multiple hosts on the Internet provides significant performance gains over a network-oblivious placement; and (2) to determine if dynamic placement based on online network monitoring provides significant performance gains over a one-time placement based on initial information. Results from this study are presented in section 5.3.

### 5.1 Variations in Internet latency

We selected 45 hosts: 15 popular .com web-sites (US), 15 popular .edu web sites (US) and 15 well-known non-US hosts. These host were pinged from four different locations in the US. The study was conducted over several weekdays, each host-pair being monitored for at least 48 hours. We used the commonly available *ping* program and sent one ping per second. This resolution was acceptable as our goal was to discover medium-term (30sec/minutes) and long-term (hours) variations.

The conclusions of our study, briefly, are: (1) there is large spatial variation in Internet latency (the per-hour mean latency varied between 15 ms and 863 ms for US hosts and between 84 ms and 4000 ms for non-US hosts); (2) there is a large and stable variation in the latency of a single host-pair over the period of a day (maximum daily variation in per-hour mean latency for US hosts was 550 ms and for non-US hosts was 5750 ms); (3) There is a lot of jitter in the latency measures but in most cases, the jitter is small; (4) there are isolated peaks in latency that appear only for a single time interval; (5) for time windows of 10 seconds or larger, the mode value (with a 10 ms jitter threshold) dominates (in most time windows, 70-90% of the latency values fall within a jitter threshold of the most common value); (6) the moving-window mode changes quite slowly.

### 5.2 Experimental Setup

Having established that there are significant spatial and temporal variations in network latency on the Internet, we

examined how well *adaptalk* could adapt to these variations.

To simulate the characteristics of long-haul networks, we decided to run our experiments over a low-latency LAN and delay all packets based on the ICMP ping traces described above (see Figure 4 (a)). This approach also allowed us to perform repeatable experiments. To ensure that delaying packets instead of using a real network does not skew the latency measures, we performed a simple test. Free-running Komodo monitors were installed at `bookworm.cs.umd.edu` and `jarlsberg.cs.wisc.edu` and were used to collect UDP latency measures between this host-pair. In parallel, a trace of ICMP ping times between these two hosts over the same period (5000 sec) was collected. This trace was later fed into trace-driven Komodo monitors running on two hosts on our LAN. The latency measures reported by the trace-driven monitors matched quite well with the actual latency measures reported by free-running monitors. The average of the actual latency measures was 128 ms (std dev = 64); the average of the values reported by the trace-driven monitors was 144 ms (std dev = 68).

We performed all our experiments on four Solaris machines on our LAN. We picked six trace-segments from the Internet study and used them to delay packets between the machines. All these segments were over the noon-2pm EDT period. We selected this period since noon is the approximate beginning of the daily latency peak for US networks as well as the approximate end of the daily latency peak for many non-US networks. These traces were selected to approximate the network latency spectrum observed in the Internet study. Hosts participating in the selected traces include: `java.sun.com`, `home.netscape.com`, `www.opentext.com`, `cesdis.gsfc.nasa.gov`, `www.monash.edu.au` and `www.ac.il`. This setup makes the four local machines behave like four far-flung machines on the Internet. Figure 4 (b) shows the configuration used for the experiments.

### 5.3 Experiments

We performed a series of experiments to evaluate the benefits of adapting to various types of network variations. The experiments consisted of running three different versions of the chat server. The first version, called static-placement, had no migration support and no network-awareness. The location of the *msgboard* was chosen in a network-oblivious fashion. The second version was a stripped-down version of *adaptalk*, called one-shot-placement. It used network information from Komodo to find the best initial placement for the *msgboard*, and used mobility support to move it there. After initial placement,

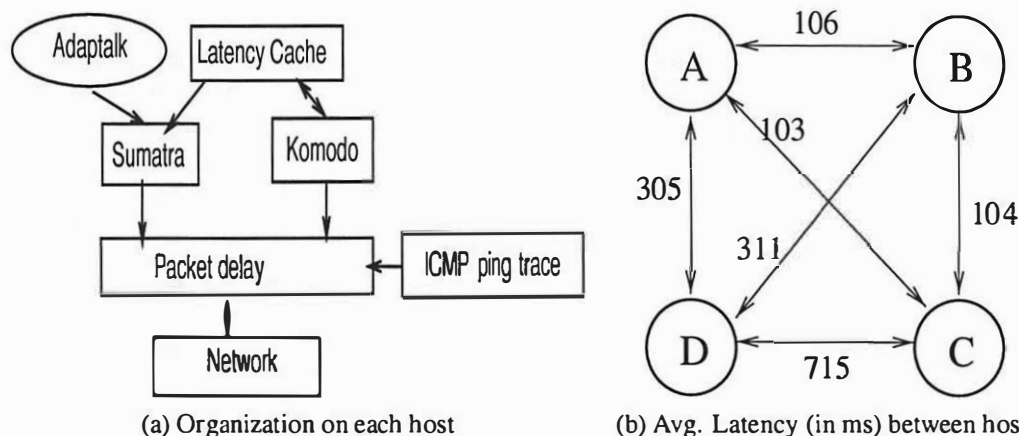


Figure 4: Experimental Setup. Four local machines on a LAN were used to simulate four remote machines on the Internet by adding delays to packets. ICMP ping traces between real Internet hosts were used to generate the delays, so as to capture real-life temporal variations in latencies.

migration decisions and network-awareness were turned off. The third version, called dynamic-placement, was the full-fledged adaptalk, as described in section 4. It used on-line monitoring and dynamic placement to position the msgboard.

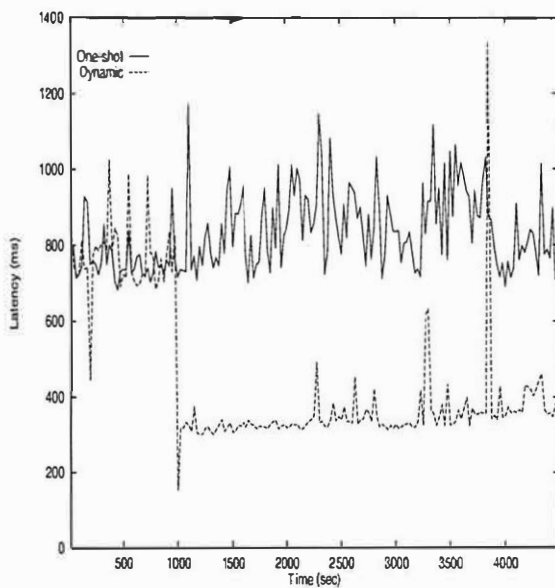
The performance of static-placement depends on the location of the msgboard. If static-placement chooses the same location as one-shot-placement, both would have the same performance. On the other hand, since static-placement is network-oblivious, it is just as likely to place the msgboard at the worst possible location. As the performance of one-shot-placement already provides a rough upper-bound on the performance of static-placement, we deliberately chose the worst initial placement when running the static-placement version.

**Adapting to Population Variation:** To evaluate the effect of changing user distribution we used the following workload: A conversation was initiated between hosts *C* and *D*. Host *B* joins the conversation after 15 minutes, and host *A* joins 15 minutes thereafter. Each host sends a sequence of 70-character sentences with a 5-second think time between sentences. With only two hosts initiating the conversation, there is no difference between the best and worst initial placements for the msgboard and both static-placement and one-shot-placement perform identically (both place the msgboard on host *D*). Figure 5 (a) plots the maximum latency over all hosts for the one-shot-placement version. Note that even after new hosts join the conversation there is no noticeable difference in maximum latency. In contrast, dynamic-placement adapts to the changing population. Soon after host *B* joins the conversation, the adaptive placement policy moves the msgboard there, causing a drop in the maximum latency.

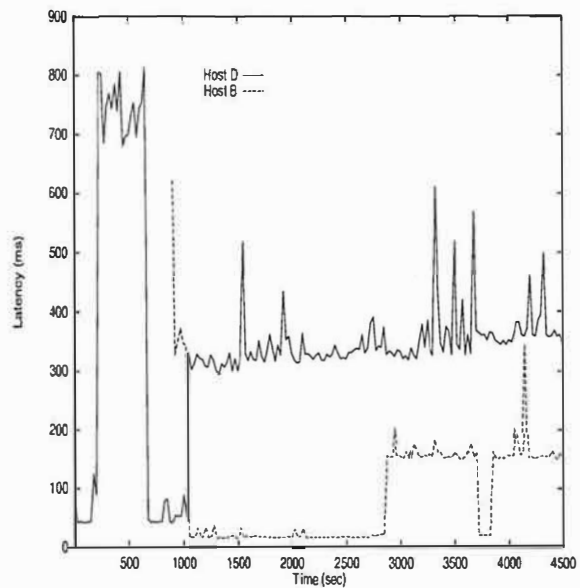
After host *A* joins the conversation, the msgboard moves between hosts *A* and *B* in response to temporal fluctuations. This can be seen from the variation in latency for host *B* in Figure 5 (b). These movements help keep the maximum latency steady even in the presence of temporal fluctuations.

**Adapting to Temporal and Spatial Variation:** In this case the client population is assumed to be stable. The workload consists of all 4 hosts jointly initiating a conversation which runs for 75 minutes. As before, each host generates a new sentence every 5 seconds. In this case, the network-oblivious (static-placement) version places the chat server on host *D*. The network-aware (one-shot-placement) version uses latency information provided by Komodo to determine that host *B* is a much better placement. For the dynamic-placement version, initial placement is less important as it should be able to recover from a bad initial placement. For this version, we place the msgboard at host *D*, the worst-possible location.

To avoid clutter, Figure 6 shows the performance of these three versions in two different graphs. Figure 6 (a) compares the maximum latency (over all participants) for the dynamic-placement and static-placement versions. As seen from the sharp drop on the left end of the graph, the dynamic-placement version is successfully able to move the msgboard away from its bad initial placement to a more suitable location. Figure 6 (b) compares the average maximum latency (over all participants) for the dynamic-placement and one-shot-placement versions. It shows that once the dynamic-placement version moves the server to a more suitable location, the performance of the two versions is largely equivalent. This implies that adapting to short-term temporal variations in a steady population work-



(a) one-shot vs. dynamic for population variation.  
Max latency (over all participants) vs time.



(b) Latency variations for hosts *B* and *D*  
Jumps signify movement of the server.

Figure 5: Adapting to population variation. Hosts *C* and *D* initiate the conversation. Host *B* joins after 900 seconds and host *A* joins 1800 seconds after the beginning. The one-shot-placement version places the chat server at host *D*. The dynamic-placement version migrates the server when new hosts join.

load does not provide much performance advantage over one-shot network-aware placement. It may, however, still be advantageous to adapt to long-term temporal variations. Note that at the far right of graph Figure 6 (b), temporal variation in the link latencies do allow the dynamic-placement version to do better than the one-shot-placement version.

## 6 Discussion

In the introduction, we raised three questions with respect to network-aware mobility. First, how should programs be structured to utilize mobility to adapt to variations in network characteristics? Second, is the variation in network characteristics such that adapting to them proves profitable? Finally, can adequate network information be provided to mobile applications at an acceptable cost?

Our experience with Sumatra and `adaptalk` provides some early insights about application structure suitable for adaptive mobile programs. First, the migration policy should be cheap so that applications don't have to analyze the tradeoffs of the migration decision itself. An easy-to-compute policy allows frequent decisions and rapid adaptation to changes in the environment. We believe that an easy-to-compute migration policy was key to `adaptalk`'s ability to quickly find good locations for the chat server.

Second, good modularization helps an application take advantage of mobility. Modularization is important for all distributed applications but it is more so for mobile programs as they have to make online decisions about the placements of different components. Third, to be resource-aware, remote accesses should be split-phase; the first phase delivers an *abbreviation*, a small and cheaply computed metric of the data (for example, size, number of data items, thumbnail sketch etc) and the second phase actually accesses the data. This allows the application to change its data access modality for the second phase (retrieve remotely, request filtering, move to data location) based on the value of the abbreviation and knowledge of its own requirements. This insight comes from our experience with writing other applications in Sumatra; `adaptalk` does not benefit from this as the size of all messages is small.

An important question that needs further investigation is where the control for mobility decisions should be placed – whether mobility decisions should be made by a central controller that keeps track of the state of all links or by multiple local controllers that use information only from a small subset of the links. Centralized decisions are likely to be more expensive than distributed decisions (the latter need less information and less synchronization) but could yield better performance (as they use global information).

To answer the second question, we evaluated the prof-

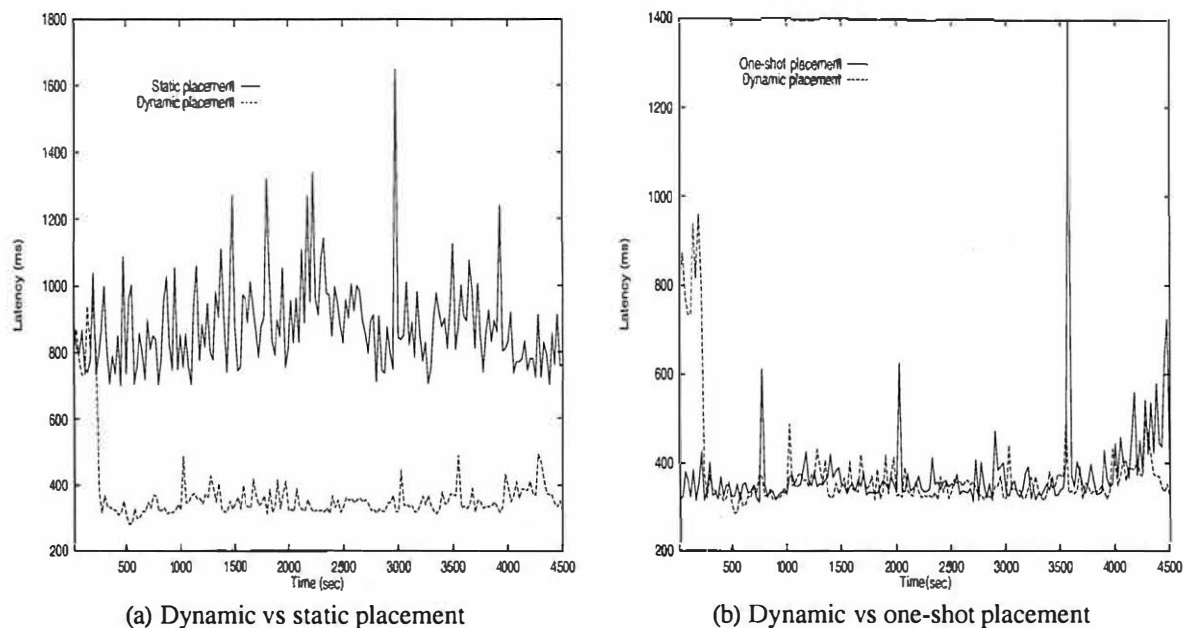


Figure 6: Maximum latency (over all participants) vs time in adaptalk. The one-shot-placement and the static-placement are computed based on latency information available when the conversation is initiated. The client population is stable throughout the experiment.

itability of adapting to changes in the user-distribution as well as spatial and temporal variations in network latency. Adapting to changes in user-distribution led to significant gains allowing adaptalk to find better placements as more users came online. Support for mobility allows applications built around a central data-structure to recover from a poor initial placement of this structure by repositioning it to a more suitable location. Adapting to temporal variations alone did not lead to significant benefits over the period of an hour. In light of this experience, we expect that a simpler migration policy for adaptalk for short periods would consider migration only when users join or leave the conversation, rather than on every message as is currently done. Since long-term variation of latency could be as large as 550 ms (US hosts) and 5750 ms (non-US-hosts), longer conversations could still benefit from adapting to temporal variations.

Our experiments with Komodo illustrate that cheap active monitoring can provide network information that can be profitably exploited. Though it would be best to use Komodo as a stand-alone system supplying network information to many distributed applications, its cost is so low that one can contemplate rolling Komodo into individual applications such as adaptalk without overloading the network. Active monitoring was needed for adaptalk as it needed information about links that are not used in the current placement but could be used in alternative placements. Other applications that change the location of com-

putation but do not change the pattern of communication would not need active monitoring as they could piggyback monitoring information on existing messages. An example of such an application would be an information access program on a mobile platform which moves primarily between this platform and a proxy host on the static network. Active monitoring, as implemented in Komodo, will not be as cheap for applications that are bandwidth-sensitive and not latency-sensitive. We are currently investigating methods to cheaply estimate Internet bandwidth.

In this paper, we have considered Internet hosts that are static. If the platform is mobile and is able to switch between multiple wireless networks [14], the temporal variation in latency could be greater and more abrupt. In these cases, adapting to short-term temporal variations could provide a significant benefit even for latency-sensitive applications.

System stability is a potential concern for programs whose components are mobile. We believe that system stability is a property of the application and not the underlying system support. Accordingly, Sumatra does not provide automatic tracking. Instead, it provides support (in the form of *object-moved* exceptions) that allows applications to track mobile objects (see section 2 for details). We have not yet encountered stability problems in any of our applications.

Finally, we would like to argue the need for mobility as an adaptation mechanism. An alternative adaptation

mechanism, which places replicated servers at all suitable points in the network, could adapt to spatial, temporal and population variation by handing off control between servers and by using dynamically created hierarchies of servers. It is quite likely that for any particular application, such a strategy would be able to achieve the performance achieved by programs that use program mobility as the adaptation tool. The advantage of mobility-based strategies is that it allows small groups of users to rapidly set up private communities on-demand without requiring extensive server placement.

## 7 Related work

Process migration and remote execution have been proposed, and have been successfully used, as mechanisms for adapting to changes in host availability [5, 7, 15, 21, 24]. Remote execution has also been proposed for efficient execution of computation that requires multiple remote accesses [6, 8, 22] and for efficient execution of graphical user interfaces which need to interact closely with the client [2]. Both these application scenarios use remote execution as a way to avoid using the network. Most proposed uses of Java [10] also use remote execution to avoid repeated client-server interaction. In these applications, decisions about the placement of computation are hardcoded. To the best of our knowledge, Sumatra (together with Komodo) is the first system that allows distributed applications to *monitor* the network state and *dynamically* place computation and data in response to changes in the network state. We also believe that our experiment with adaptalk is the first attempt to determine if the variation in Internet characteristics is such that it is profitable for applications to adapt to them.

Network-awareness is particularly important to applications running on mobile platforms which can see rapid changes in network quality. Various forms of network-awareness have been proposed for such applications. Application-transparent or system-level adaptation to variations in network bandwidth has been successfully used by the designers of the Coda file system [17] to improve the performance of applications. The Odyssey project on mobile information access plans to provide support for application-specific resource monitoring and adaptation. The primary adaptation mechanism under consideration is change in data fidelity [20]. Athan and Duchamp [1] propose the use of remote execution for reducing the communication between a mobile machine and the static network. In all these systems, location of the various computation modules is fixed; adaptation is achieved by changing the way in which the network is used.

Several systems have been built which permit an executing program to move while it is in execution - for example Obliq [3], Agent TCL [11], Emerald [13], Telescript [23] and TACOMA [12]. The primary distinction between these systems and Sumatra is that in Sumatra, *all* communication and migration happens under application control. Complete application control allows us to easily explore different policy alternatives for resource monitoring and for adapting to variations in resources.

Several studies have been performed to determine end-to-end Internet performance. Sanghi et al [19] and Mukherjee [16] have studied network latency. Their observations show that while round trip times show significant variability with sharp peaks, there exist dominant low frequency components. This is consistent with our observations that in a time window of reasonable size, the mode value usually dominates and that the mode value changes slowly.

Golding [9] and Carter and Crovella [4] have studied mechanisms to estimate end-to-end Internet bandwidth. Golding's results indicate that attempts to predict bandwidth using previous observations alone is unlikely to work well. Carter and Crovella propose the use of round trip times for short packets to estimate network congestion. They propose to use the network congestion information to estimate changes in network bandwidth (assuming the inherent bandwidth of the link has been previously computed by flooding the link). Their results indicate that it might be possible to estimate the *change* in network bandwidth using information about the *change* in network latency.

## 8 Conclusions and future work

This paper is a first step in demonstrating that distributed programs can use mobility as a tool to adapt to variations in their operating environment. Our exploration of network-aware mobile programs lead us to the following conclusions. First, network-aware placement of components of a distributed application can provide significant performance gains over a network-oblivious placement. For short term applications (applications that run for an hour or so), exploiting spatial variations as well as variations in the number and location of the clients achieves most of the gains. For longer-running applications, exploiting temporal variations might be worthwhile. Second, effective mobility decisions can be based on coarse-grained monitoring. This allows cheap active monitoring without losing effectiveness. Finally, there is significant spatial and temporal variation in Internet latency which can be effectively adapted to by mobile programs.

We believe that there is a class of long running applications over the Internet for which resource-aware mobility could provide flexibility and performance which would

take a lot more effort to achieve by other means. One future direction we would like to pursue is to identify such applications and understand their structure and requirements. Some of the examples we intend to study include network-bandwidth-aware data-combination on the Internet, custom combination of periodically generated large-volume datasets (such as weather information) and resource-aware pre-fetching for web clients.

Another direction that we plan to explore is efficient distributed monitoring of other resources, in particular network bandwidth and server availability. We are investigating cheap methods of estimating network bandwidth. An important question that we are investigating is how accurate resource estimates need to be in order to benefit from resource aware mobility and how the accuracy of estimation affects performance.

Network-awareness is very important to applications running on mobile platforms which can see rapid changes in network quality. The nature of these variations is significantly different from those that we have seen in our Internet study. We are planning to extend our work to mobile platforms. In particular, we would like to understand the monitoring requirements and the mobility algorithms in a mobile computing environment.

The focus of our work has been to make distributed applications achieve better performance using mobility as a tool to adapt to resource variations. We have therefore not as yet addressed the important issues of security and resource-use containment in our implementation. We plan to look into both these issues.

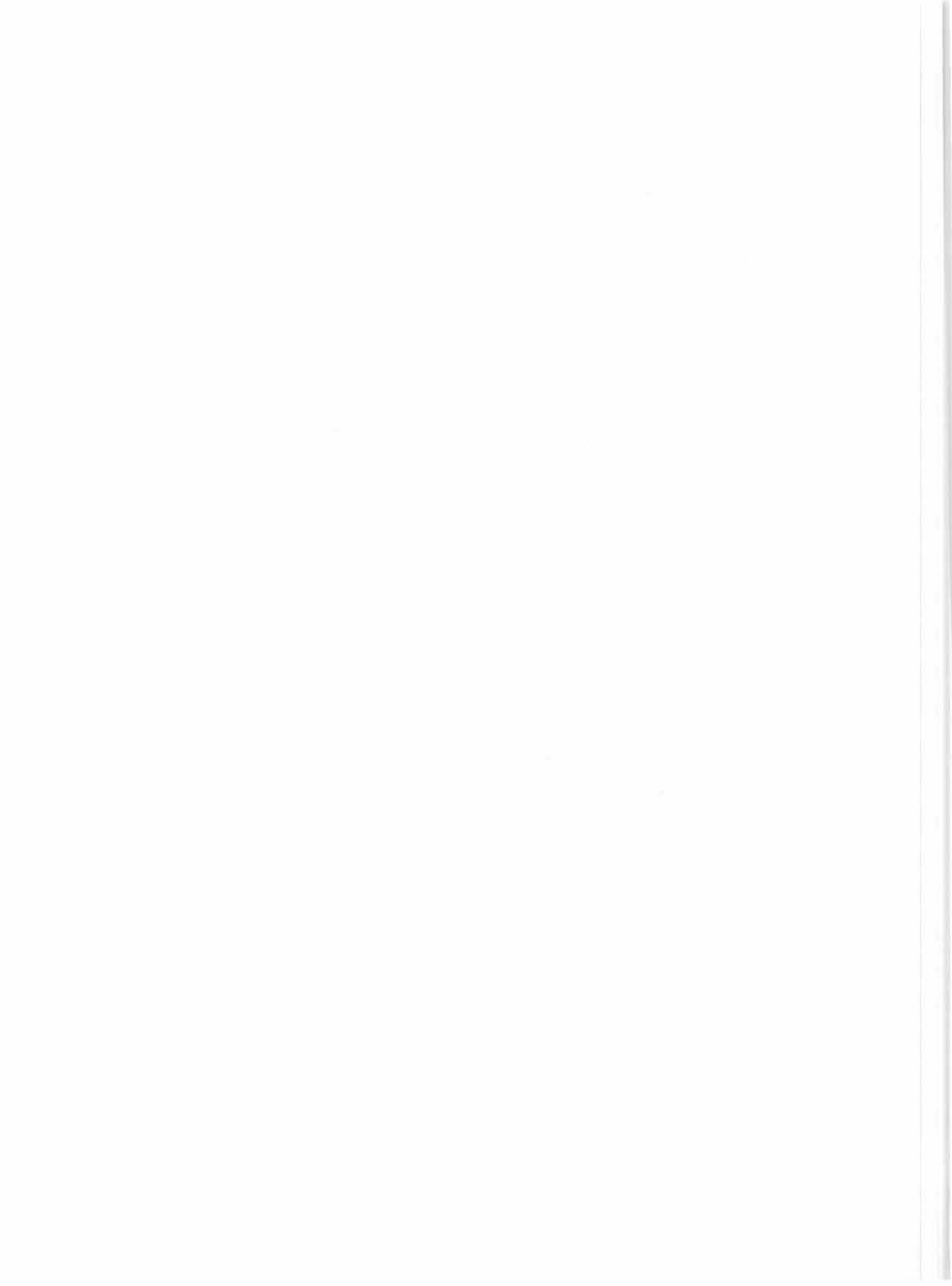
## Acknowledgments

We would like to thank Mustafa Uysal, Manuel Ujaldon and anonymous referees for their suggestions. We would also like to thank John Kohl, our shepherd.

## References

- [1] A. Athan and D. Duchamp. Agent-mediated Message Passing for Constrained Environments. In *Proceedings of the USENIX Mobile and Location-independent Computing Symposium*, pages 103–7, Aug 1993.
- [2] K. Bharat and L. Cardelli. Migratory Applications. In *Proceedings of the Eighth ACM Symposium on User Interface Software and Technology*, pages 133–42, Nov 1995.
- [3] L. Cardelli. A Language with Distributed Scope. In *Proceedings of the 22<sup>nd</sup> ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1995.
- [4] R. Carter and M. Crovella. Dynamic Server Selection using Bandwidth Probing in Wide-Area networks. Technical Report BU-CS-96-007, Boston University, March 1996.
- [5] J. Casas, D. Clark, R. Konuru, S. Otto, and R. Prouty. MPVM: A Migration Transparent Version of PVM. *Computing Systems*, 8(2):171–216, Spring 1995.
- [6] S. Clamen, L. Leibengood, S. Nettles, and J. Wing. Reliable Distributed Computing with Avalon/Common Lisp. In *Proceedings of the International Conference on Computer Languages*, pages 169–79, 1990.
- [7] F. Douglass and J. Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software - Practice and Experience*, 21(8):757–85, Aug 1991.
- [8] J. Falcone. A Programmable Interface Language for Heterogeneous Systems. *ACM Transactions on Computer Systems*, 5(4):330–51, November 1987.
- [9] R. Golding. End-to-end performance prediction for the Internet (Work In Progress). Technical Report UCSC-CRL-92-26, University of California at Santa Cruz, June 1992.
- [10] J. Gosling and H. McGilton. The Java Language Environment White Paper, 1995.
- [11] R. Gray. Agent TCL: A Flexible and Secure Mobile-agent System. In *Proceedings of the Fourth Annual Tcl/Tk Workshop (TCL 96)*, July 1996.
- [12] D. Johansen, R. van Renesse, and F. Schneider. An Introduction to the TACOMA Distributed System Version 1.0. Technical Report 95-23, University of Tromso, 1995.
- [13] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(2):109–33, February 1988.
- [14] R. Katz. The Case for Wireless Overlay Networks. Invited talk at the ACM Federated Computer Science Research Conferences, Philadelphia, 1996.
- [15] M. Litzkow and M. Livny. Experiences with the Condor Distributed Batch System. In *Proceedings of the IEEE Workshop on Experimental Distributed Systems*, Huntsville, Al., 1990.

- [16] A. Mukherjee. On the dynamics and significance of low frequency components of Internet load. *Internet-working: Research and Experience*, 5(4):163–205, Dec 1994.
- [17] L. Mummert, M. Ebling, and M. Satyanarayanan. Exploiting Weak Connectivity for Mobile File Access. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, December 1995.
- [18] M. Ranganathan, A. Acharya, and J. Saltz. Distributed Resource Monitors for Mobile Objects. In *Proceedings of the Fifth International Workshop on Operating System Support for Object Oriented Systems*, pages 19–23, October 1996.
- [19] D. Sanghi, A.K. Agrawala, O. Gudmundsson, and B.N. Jain. Experimental Assessment of End-to End Behavior on Internet. Technical Report CS-TR-2909, University of Maryland, June 1992.
- [20] M. Satyanarayanan, B. Noble, P. Kumar, and M. Price. Application-aware adaptation for mobile computing. *Operating Systems Review*, 29(1):52–5, Jan 1995.
- [21] J. Smith. A Survey of Process Migration Mechanisms. *Operating Systems Review*, 22(3):28–40, July 1988.
- [22] J. Stamos and D. Glifford. Implementing Remote Evaluation. *IEEE Transactions on Software Engineering*, 16(7):710–22, July 1990.
- [23] J. White. Telescript Technology: Mobile Agents. <http://www.genmagic.com/Telescript/Whitepapers>.
- [24] E. Zayas. Attacking the Process Migration Bottleneck. In *Proceedings of the Eleventh ACM Symposium on Operating System Principles*, pages 13–24, November 1987.



# Using Smart Clients to Build Scalable Services

Chad Yoshikawa, Brent Chun, Paul Eastham,  
Amin Vahdat, Thomas Anderson, and David Culler  
*Computer Science Division  
University of California  
Berkeley, CA 94720*

## Abstract

Individual machines are no longer sufficient to handle the offered load to many Internet sites. To use multiple machines for scalable performance, load balancing, fault transparency, and backward compatibility with URL naming must be addressed. A number of approaches have been developed to provide transparent access to multi-server Internet services including HTTP redirect, DNS aliasing, Magic Routers, and Active Networks. Recently however, portable Java code and lightly loaded client machines allow the migration of certain service functionality onto the client. In this paper, we argue that in many instances, a client-side approach to providing transparent access to Internet services provides increased flexibility and performance over the existing solutions. We describe the design and implementation of *Smart Clients* and show how our system can be used to provide transparent access to scalable and/or highly available network services, including prototypes for: telnet, FTP, and an Internet chat application.

## 1 Introduction

The explosive growth of the World Wide Web is straining the architecture of many Internet sites. Slow response times, network congestion, and “hot sites

of the day” being overrun by millions of requests are fairly commonplace. These problems will only worsen as the Web continues to experience rapid growth. As a result, it has become increasingly important to design and implement network services, such as HTTP, FTP, and web searching services, to scale gracefully with offered load. Such scalable services must, at minimum, address the following issues:

- **Incremental Scalability** – If the offered load begins to exceed a service’s hardware capacity, it should be a simple operation to add hardware resources to transparently increase system capacity. Further, a service should be able to recruit resources to handle peaks in the load. For example, while the US Geological Survey Web site (<http://quake.usgs.gov>) is normally quite responsive, it was left completely inaccessible immediately after a recent San Francisco Bay Area earthquake.
- **Load Balancing** – Load should be spread dynamically among server resources so that clients receive the best available quality of service.
- **Fault Transparency** – When possible, the service should remain available in the face of server and network upgrades or failures.
- **Wide Area Service Topology** – Individual servers comprising a service are increasingly distributed across the wide area [Net 1994, Dig 1995]. The server machines that comprise a network service should not be required to have a restricted or static topology. In other words, all servers should be allowed to arbitrarily migrate to other machines.
- **Scalable Service To Legacy Servers** – Adding scalability to existing network services such as FTP, Telnet, or HTTP should not require modifications to existing server code.

---

This work was supported in part by the Defense Advanced Research Projects Agency (N00600-93-C-2481, F30602-95-C-0014), the National Science Foundation (CDA 0401156), California MICRO, the AT&T Foundation, Digital Equipment Corporation, Exabyte, Hewlett Packard, Intel, IBM, Microsoft, Mitsubishi, Siemens Corporation, Sun Microsystems, and Xerox. Anderson was also supported by a National Science Foundation Presidential Faculty Fellowship. Yoshikawa is supported by a National Science Foundation Fellowship. The authors can be contacted at {chad, bnc, eastham, vahdat, tea, culler}@cs.berkeley.edu.

Unfortunately, providing these properties for network services while remaining compatible with the de facto URL (Uniform Resource Locator) naming scheme has proven difficult. URLs are by definition location dependent and hence are a single point of failure and congestion. A number of efforts address this limitation by hiding the physical location of a particular service behind a logical DNS hostname. Examples of such systems include HTTP redirect, DNS Aliasing, Failsafe TCP, Active Networks, and Magic Routers.

We argue that in many cases the client, rather than the server, is the right place to implement transparent access to network services. We will describe limitations associated with each of the above solutions and demonstrate how these limitations can be avoided by moving portions of server functionality onto the client machine. This approach offers the advantage of increased flexibility. For example, clients aware of the relative load on a number of FTP mirror sites can connect to the least loaded mirror to deliver the highest throughput to the end user. Ideally, the selection and connection process takes place without any intervention from the end user, unlike the Web today where users must choose among FTP mirror sites manually. Note that in this example, clients must take into account available network bandwidth to each mirror site as well as the relative load of the sites to receive optimal performance. Such flexibility would be difficult to provide with existing server-side solutions since individual servers may not have knowledge of mirror site group membership and client location.

The migration of service functionality onto client machines is enabled by two recent developments. Today, most popular Internet services, such as FTP, HTTP, and search engines are universally accessed through extensible Web browsers. This extensibility allows insertion of service-specific code onto client machines. In addition, the advent of Java [Gosling & McGilton 1995] allows such code to be easily distributed to multiple platforms. Next, network latency and bandwidth are increasingly the bottleneck to the performance delivered to clients. Thus, client processors can be left relatively idle. We will demonstrate that offloading service functionality onto these idle cycles can substantially improve the quality of service along the axis described above.

Motivated by the above observations, we describe the design and implementation of *Smart Clients* to support our argument for client-side location of code for scalability and transparency. The central idea behind Smart Clients is migrating server functionality to the client machine to improve the overall quality of service in the ways described above. This ap-

proach contrasts the traditional "thin-client" model where clients are responsible largely for displaying the results of server operations. While our approach is general, this paper concentrates on augmenting the client-side architecture to provide benefits such as fault transparency and load balancing to the end user.

The rest of this paper is organized as follows. Section 2 discusses existing solutions to providing scalable services. The limitations of the existing solutions motivates the Smart Client architecture, described in Section 3. Section 4 demonstrates the utility of the architecture by describing the implementation and performance of interfaces for telnet, FTP, and a scalable chat service. Section 5 evaluates our requirements above in the context of the Smart Client architecture. Section 6 describes related work, and Section 7 concludes.

## 2 Alternative Solutions

Existing architectures include DNS Aliasing [Brisco 1995, Katz et al. 1994], HTTP redirect [Berners-Lee 1995], Magic Routers [Anderson et al. 1996], failsafe TCP [Goldstein & Dale 1995], and Active Networks [Wetherall & Tennenhouse 1995]. Figure 1 describes how Smart Clients fits in the space of existing solutions. We will describe each of the existing solutions in turn leading to a description of the Smart Client architecture.

A number of Web servers use Domain Name Server (DNS) aliasing to distribute load across a number of machines cooperating to provide a service. A single logical hostname for the service is mapped onto multiple IP addresses, representing each of the physical machines comprising the service. When a client resolves a hostname, alternative IP addresses are provided in a round-robin fashion. DNS aliasing has been demonstrated to show relatively good load balancing, however the approach also has a number of disadvantages. First, random load balancing will not work as well for requests demonstrating wide variance in processing time. Second, DNS aliasing cannot account for geographic load balancing since DNS does not possess knowledge of client location/server capabilities.

On a client request, HTTP redirect allows a server to instruct the client to send the request to another location instead of returning the requested data. Thus, a server machine can perform load balancing among a number of slave machines. However, this approach has a number of limitations: latency to the client is doubled for small requests, a single point of failure is still present (if the machine serving redirects is un-

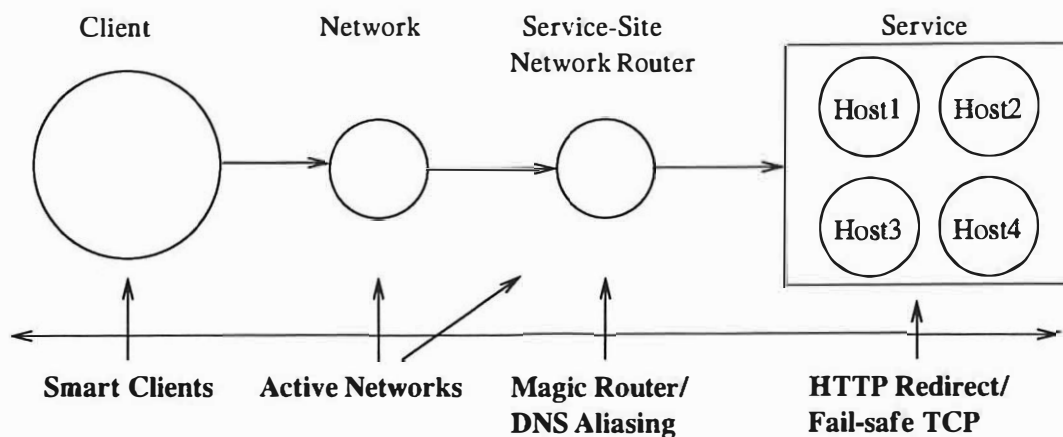


Figure 1: This figure describes the design space for providing transparent access to scalable network services. Transparency mechanisms can be implemented in a number of places, including the client, network, network routers, or at the service site.

available, the entire service appears unavailable), and servers can still be overloaded attempting to serve redirects. Further, this mechanism is currently only available for HTTP; it does not work with legacy services nor does it optimize wide-area access.

The Magic Router provides transparent access by placing a modified router on a separate subnet from machines implementing a service. The Magic Router inspects and possibly modifies all IP packets before routing the packets to their destination. Thus, it can perform load balancing and fault transparency by mapping a logical IP address to multiple server machines. If a packet is destined for the designated service IP address, the Magic Router can dynamically modify the packet to be sent to an alternative host. Unresolved questions with Magic Routers include how much load can be handled by the router machine before the dynamic redirection of the packets becomes the bottleneck (since it must process every packet destined for a particular subnet). Magic Routers also require a special network topology which may not be feasible in all situations. Finally, the Magic Router is not aware of the load metrics relevant to individual services, i.e. it would have to perform remappings based on a generic notion of load such as CPU utilization.

Fail-safe TCP replicates TCP state across two independent machines. On a server failure, the peer machine can transparently take over for the failed machine. In this fashion, fail-safe TCP provides fault transparency. However, it requires a dedicated backup machine to mirror the primary server, and it does not address the problem of the front-end becoming a bottleneck. Finally, both fail-safe TCP and

Magic Routers are relatively heavy-weight solutions requiring extra hardware.

Proposals for Active Networks allow for computation to take place in network routers as packets are routed to their destination. This approach could potentially provide fault transparency and load balancing functionality inside of the routers. We believe Active Networks, if widely deployed, can provide a mechanism for implementing Smart Client functionality.

All of the above solutions provide a level of transparent access to network services with respect to load balancing and fault transparency. However, they are all limited by the fact that they are divorced from the characteristics and implementations of individual services. We observe that the greatest functionality and flexibility can often be provided by adding service-specific customization to the client, rather than service-independent functionality on the server.

### 3 Smart Client Architecture

In this section, we describe how the Smart Client architecture allows for the construction of scalable services. For the purposes of this paper, we assume the service is implemented by a number of peer servers, each capable of handling individual client requests<sup>1</sup>. The key idea behind Smart Clients is the migration of certain server functionality and state to the client machine. This approach provides a number of advantages: (i) offloading server load and decreasing imple-

<sup>1</sup>This assumption holds for many widely-used Internet services such as HTTP, FTP, and Web searching services.

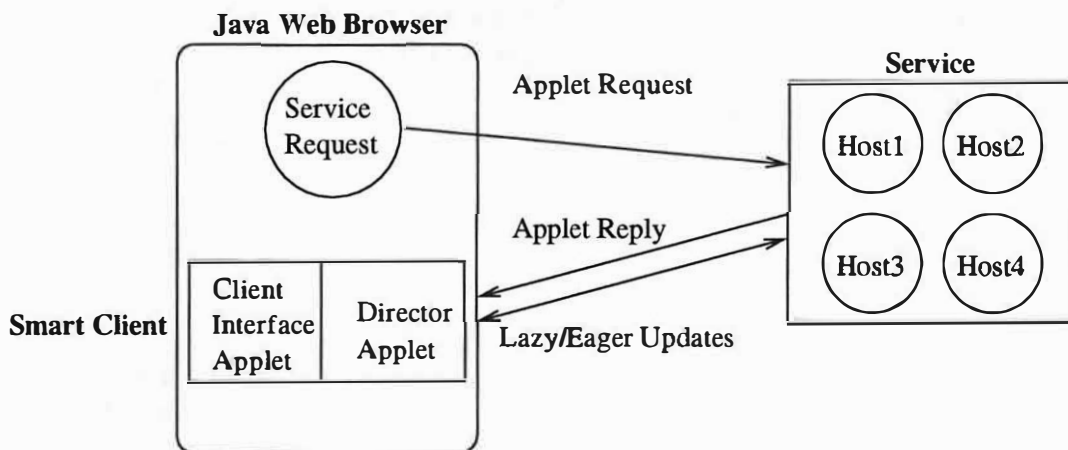


Figure 2: This Figure describes the Smart Client service access model. Two service-specific Java applets are supplied to mediate server access. The *client interface* applet provides the interface to the user and makes requests of the service. The *director* applet is responsible for providing transparency to the client applet; it makes server requests to the appropriate (e.g. least loaded) server, and updates its notion of server state.

mentation complexity, (ii) allowing clients to utilize multiple peer servers distributed across the wide area without the knowledge of individual servers, and (iii) improving the load distribution and fault transparency of the service as a whole.

When a user wishes to use a service, a bootstrapping mechanism is used to retrieve service-specific applets designed to access the service. Two cooperating applets, a *client interface applet* and a *director applet*, provide the interface and mask the details of contacting individual servers respectively. Client-side functionality is partitioned in this fashion to separate the service's interface design from the mechanisms necessary to deliver client requests to servers in a load-balanced, fault tolerant manner.

The client interface applet is responsible for accepting user input and packaging these requests to the director applet. The director applet encapsulates knowledge of the service member set and the service-specific meta-information allowing the director applet to send requests to the appropriate server. For every user request, the Smart Client uses the director applet to invoke a service-specific mechanism for determining the correct destination server for the request. Figure 3 shows the interaction of the two applets in a Java-enabled Web browser. A number of issues are associated with this approach: naming mechanisms for choosing among machines implementing a service, procedures for receiving updates with new information about a service (e.g., changes in load, or the availability of a new machine), and bootstrapping retrieval of the Smart Client applets. We will discuss

each of these issues in turn leading to a description of the Smart Clients API.

### 3.1 Transparent Service Access

#### 3.1.1 Load Balancing and Fault Transparency

We begin our discussion of the Smart Client architecture by describing the techniques used to provide load balanced and fault tolerant access to network services. Discussion of bootstrapping the retrieval of the Smart Client is deferred until Section 3.2. We assume that services accessed by Smart Clients are implemented by a number of peer servers. In other words, any of a list of machines are capable of serving individual client requests. Thus, the director applet makes a service-specific choice of a physical host to contact based on an internal list of (dynamically changing) server sites. Ideally, this choice should balance load among servers while maximizing performance to the end user.

While the choice of load balancing algorithm is service specific, we enumerate a number of sample techniques. The simplest approach is to randomly pick among service providers. While this approach is simple to implement and does not require server modifications, it can result in both poor load balancing and poor performance to the end user. For example, an FTP applet picking randomly among a list of service providers may pick an under-powered mirror site on another continent. A refinement on random load balancing would bias future random choices based on how quickly requests to a particular server are pro-

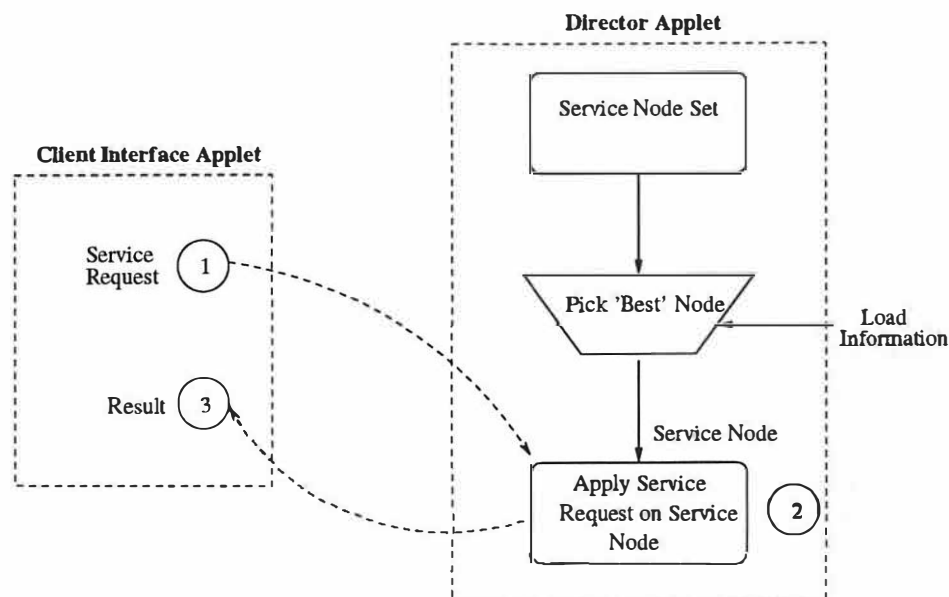


Figure 3: This Figure describes the Smart Client architecture. (1) The client interface applet first makes a request to the director applet. (2) The director applet, given outside information such as load and changes to the service group membership, picks the best node to apply the request to. The director will also re-apply the request if the operation fails. (3) The result of the operation, including a success/failure code, is returned to the client interface applet.

cessed. For services where multiple successive requests are likely, we believe this technique should result in good performance while maintaining implementation simplicity.

Another technique involves maintaining service-specific profiles of servers. In the FTP example above, a description of hardware performance and network connectivity (perhaps using techniques similar to the Internet Weather Report [Mat 1996]) may be associated with each server. The Smart Client director applet can then use this information to evaluate available bandwidth to each server based on the client's location. A further improvement requires maintaining load information for each server. In this case, the client is able to maximize performance by weighing a combination of network connectivity, server performance, and current server load.

The mechanisms used for load balancing can be adapted to provide fault transparency to the end user. Techniques such as keep-alive messages or time outs can be utilized to determine server failure. Upon failure, the director applet can reinvoke the load balancing mechanism to choose an alternate server and reapply the request. By storing all uncompleted server requests and necessary client state information, the director applet can connect to an alternative site to retransmit all outstanding requests transparently to the

user.

### 3.1.2 Updating Applet State

In order to make load balancing decisions, the client may need a reasonably current profile of the individual servers providing the service. Depending on the application, updating the director of changes in service state can be achieved through either lazy or eager techniques, presenting both performance and semantic tradeoffs for maintaining consistency.

Examples of eager update techniques include client polling and server callbacks. Using client polling of servers to maintain load information has the disadvantage of severely loading server machines. Server callback techniques can be more scalable than client polling, however they require server modifications and increase implementation complexity. Neither client polling nor server callbacks is likely to scale to the level of thousands of clients necessary for some Web services. Eager update methods are appropriate when accurate information is required and the scale of the service is small enough to support eager protocols.

Lazy update techniques [Ladin et al. 1992] are likely to be more appropriate in the context of the Web. Lazy updates reduce network traffic by sending information only occasionally, after a number of up-

dates have been collected. One particularly attractive mode of lazy updates is piggy-backing update information with server replies to client requests. For example, a server can inform Smart Client directors applets of the addition of new server machines comprising the service when replying to a director request.

### 3.1.3 Director Architecture

Smart Clients provide a very flexible mechanism for implementing service-specific transparency. The Smart Client director provides the illusion of a single, highly-available machine to the programmer of the client interface applet. Requests made by the Smart Client client interface applet are written to operate on a single machine. The director applet chooses the destination server based on service-specific information such as load, availability, processor speed, or connection speed.

The director accepts arbitrary requests of the form "perform this action on a server node". The director applet sends the request to the server determined to deliver the best performance to the client. If the request fails, the next server in the director applet's ranked list is contacted with the request. In this way, the director applet provides transparent access to arbitrary server groups. As a result of the well-defined interface between the client interface applet and the director applet (as described in Section 3.3), individual director applets are easily interchanged for many different services. For example, the director applet providing transparent Telnet access to a cluster of workstations can also be used for services such as chat or FTP.

## 3.2 Bootstrapping Applet Retrieval

The goal of transparent access to network services would be compromised if the Smart Client applets necessary for service access must be downloaded from a single hostname before every service access. We have created a scalable bootstrapping mechanism to circumvent this single point of failure. To remove the single point of failure associated with a single hostname, we have modified jfox [Wendt 1996], an existing Java Web browser, to support a new *service* name space, e.g. *service://now chat service*. For the servicename space, the browser contacts one of many well-known search engines with a query. These well-known search engines serve the same purpose as the root name servers in DNS.

Currently, the browser contacts Altavista [Dig 1995] with a query requesting an HTML page whose title matches the service name, e.g. "now chat ser-

vice". In this way, Smart Clients leverages highly-available search engines to provide translations from well-known service names to a URL. The URL points to a page containing a *service certificate*. The certificate includes references to both the client interface and director applets. In addition, the certificate contains some initial guess as to service group membership. This hint initializes the director applet, allowing the applet to validate the list by contacting one of the nodes. Figure 4 shows a certificate used for the NOW chat service.

Jfox has been extended to cache the Smart Client applets associated with individual services, the location of the service certificate, the certificate itself and any additional state that the Smart Client director needs for the next access to the service. While the client interface applet and service-certificate are cached using normal browser disk caching mechanisms, the director state is saved by serializing the director applet (and any relevant instance variables) to disk using Java Object serialization [Jav 1996]. Thus, on subsequent service accesses, the director applet need not rely on the initial group membership contained in the service certificate. Instead, it can use the last known service group membership. With this bootstrapping mechanism, no network communication is necessary to load the service applets after the initial access.

Currently, the service certificate and applets are cached indefinitely. In the future, we plan on adding a time-out period to the server certificate. After the timeout, the browser can revalidate both its service certificate and the associated applets. If either the certificate or applets are inaccessible, the decision to proceed with the cached state can be made on a service-specific basis.

Note that with the exception of bootstrapping, the implemented applets work on unmodified Java-capable Web browsers such as Netscape Navigator and Internet Explorer. Further, mainstream browsers such as Internet Explorer allow for installation of filters over the entire browser [Leach 1996]. Such a filter would allow our bootstrapping mechanism to be implemented in widely used Web browsers.

The bootstrapping problem has been addressed in other contexts. For example, distributed applications need access to DNS without a name server. Such applications fall back to sending queries well-known root name servers when it is unable to resolve a hostname. As another example, applications which communicate through RPCs must bind to a server without using an RPC. This problem is also addressed by using broadcast to initiate binding to RPC servers on the network.

```

<HTML>
<TITLE>now chat service</TITLE>
<META name="description" content="now chat service">
<META name="keywords" content="now chat service">
<APPLET name="now_chat"
        codebase="Chat" code="Chat.class">
<param name="director" value="now_chat_director">
</APPLET>
<APPLET name="now_chat_director"
        codebase="Chat" code="ChatDirector.class">
<param name="nodes"
        value="u81.cs.berkeley.edu, u82.cs.berkeley.edu, u83.cs.berkeley.edu">
</APPLET>
</HTML>

```

Figure 4: This example of a *service certificate* references both the client interface applet (Chat.class) and the director applet (ChatDirector.class). Initial service group membership (u81,u82 and u83) is fed to the director applet for bootstrapping purposes. The director applet contacts one of these machines to obtain group membership updates.

### 3.3 Smart Clients API

In this subsection, we will describe the Smart Clients API. The goal of the API is to provide a generic interface for service providers to develop transparent access to their servers and to make it easier for programmers to implement applications for distributed services. In the interests of brevity, we do not document the interface in its entirety. Interested readers can download the Java classes implementing the API to see how the classes are used to implement a number of sample applications (as described in Section 4).

Figure 5 presents a high-level overview of the Java methods which make up the Smart Clients API. The IDirector interface provides a simple abstraction of a service to the application programmer. The programmer makes director requests through the IDirector interface. The requests are then sent by the director applet to one of the service nodes; note that the application programmer is not concerned with managing server nodes. If the request fails, a director exception is raised. In response, the director will first allow the request to clean up any state, then resend the request to another server. The director applet takes a best effort approach in delivering the request. Thus, a return of false from the delivery request indicates a catastrophic failure of the service, i.e. all servers have failed.

## 4 Sample Applications

### 4.1 Telnet Front-End for a NOW

The NOW (Network of Workstations) Project [Anderson et al. 1995a] at UC Berkeley provides approximately 100 workstations for use within the depart-

ment; however, it is difficult for users to know which of the 100 machines is least loaded. To address this problem, we developed a Web page containing a single button which, when pressed, opens a telnet window onto the least loaded machine in the NOW cluster.

The implementation of the telnet application is straightforward. The telnet Web page encapsulates the necessary Smart Clients applets. The director applet periodically polls the NOW's operating system, GLUnix [Ghormley et al. 1995], to retrieve the load averages of machines in the cluster through a simple Common Gateway Interface (CGI) program. When the user clicks on the telnet button (provided by the client interface applet), a request is sent to start a telnet window on the least loaded machine in the cluster. If the director applet notices that a machine has failed it will not submit telnet requests to that node. We are currently investigating a fault-tolerant telnet service which re-opens a telnet window (with saved state such as the current working directory and environment variables) in the event of node failure. The fault-tolerant telnet would pass this saved state through the RequestException object (as described in Figure 5).

### 4.2 Scalable FTP Interface

We have also used Smart Clients to build a scalable frontend for FTP sites. As a motivating example, the Netscape Navigator FTP download page<sup>2</sup> contains twelve hyperlinks for netscape FTP hosts. Users choose among netscape sites or mirrors to perform

<sup>2</sup>[http://www.netscape.com/comprod/mirror-client\\_download.html](http://www.netscape.com/comprod/mirror-client_download.html)

```

// Interface to encapsulate all client interface applet requests
public interface IRequest {
    // Downcall from the director to the request object. Perform the action
    // on 'hostname'. Throw RequestException if an error occurs
    public void action(String hostname) throws RequestException;
    // Downcall from the director to the request object upon failure. Perform
    // any necessary cleanup code. The state of the failed request consists
    // of the 'oldhostname' and the RequestException that was thrown from the
    // action method
    public void cleanup(String oldhostname, RequestException t);
}

// Generic interface for all director applets
public interface IDirector {
    // Execute the request r on a hostname of the director's choosing. If the
    // request object throws a RequestException, assume failure of the node
    // and reapply the request after calling the request's cleanup method.
    // If there are no remaining nodes, return false. Otherwise, return true.
    public boolean apply(IRequest r);
}

```

Figure 5: This Figure describes some of the interfaces in the Smart Clients API. Classes that implement the director interface have been written to provide much of the functionality necessary to simple directors, including randomized directors (picking a random machine) and directors based on choosing the least loaded server.

manual load balancing. To improve on this interface, Smart Client applets present a single download button to the user. The client interface applet delivers requests to the director to retrieve a file, while the director picks a machine at random from a static set of nodes. When the user presses the button, the applet transparently determines the best site for file retrieval.

To demonstrate the scalability available from using Smart Clients, we measure delivered bandwidth to Smart Client applets running in Netscape Navigator from a varying number of FTP servers. We emphasize that the choice of FTP site is transparent to the end user (a single button is pressed to begin file retrieval) and that our FTP application can be downloaded to run with unmodified servers and Java-compliant browsers. The tests were run on a cluster of Sun Sparcstation 10's and 20's interconnected by a 10 Mbps Ethernet switch. The Ethernet switch allows each machine in the cluster to simultaneously deliver 10 Mb of aggregate bandwidth to the rest of the cluster without the contention associated with shared Ethernet networks. Either one, two, or four of the machines are designated FTP servers, while the rest of the machines in the cluster attempt 40 consecutive retrievals of a 512 KB file. This experimental setup approximates multiple FTP mirror sites spread across the wide area.

Figure 6 summarizes the results of the FTP scalability tests. The graph shows aggregate delivered

bandwidth in megabytes per second as a function of the number of client machines making simultaneous file requests. For one FTP server, 8 clients are able to saturate the single available Ethernet link at 1.2 MB/s<sup>3</sup>. The results for two and four FTP servers demonstrate that the random selection of an FTP server used within the applet delivers reasonable scalability. Sixteen clients are able to retrieve approximately 2 MB/s from two servers, while 16 clients saturate four servers at approximately 3 MB/s.

For small number of clients, a single FTP server demonstrates the best performance because all 40 file downloads are made during a single connection. For the multi-server tests, multiple connections and disconnections take place as the clients attempt to randomly balance load across the servers. In the future, this problem can be avoided by implementing site affinity with successive file requests (if delivered bandwidth on the previous was deemed satisfactory), implementing a load daemon on the nodes to allow the clients to continuously choose lightly loaded machines, or by using services such as the Internet Weather Map [Mat 1996] to choose low-latency hosts. This information can be used to incrementally scale connections to available FTP servers (i.e. allow some machines to be recruited only when needed).

<sup>3</sup>We were unable to take measurements for more than 16 simultaneous clients making requests to a single server because the FTP server would not allow more than 16 simultaneous file retrievals. We plan to investigate this limitation further.

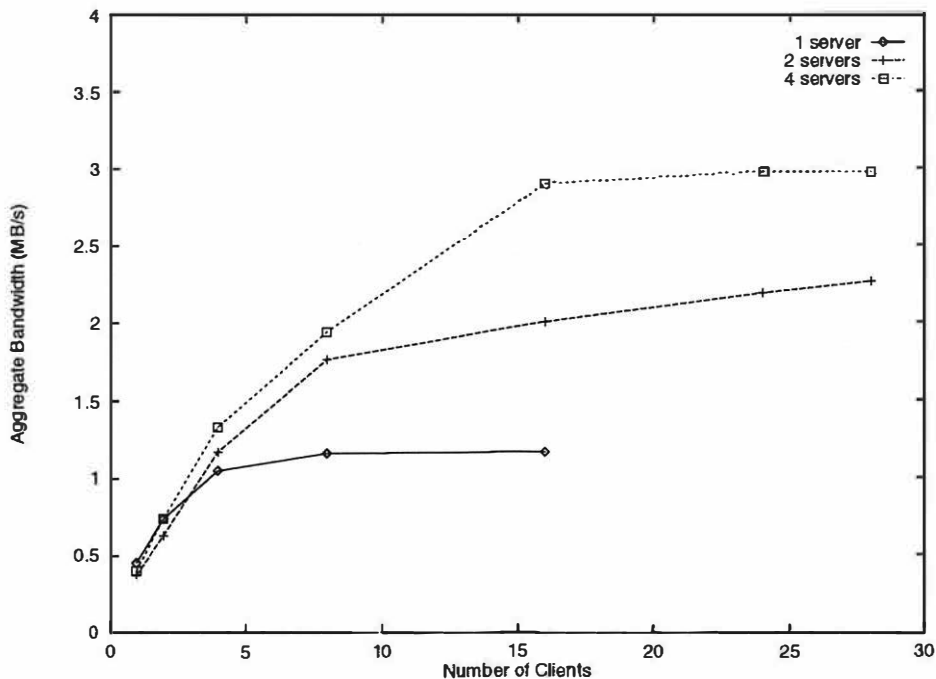


Figure 6: This figure demonstrates how a Smart Client interface to FTP delivers scalable performance. The graph shows delivered aggregate bandwidth as a function of number of clients making simultaneous requests.

### 4.3 Scalable Chat

The next application we implement is Internet chat. The application allows for individuals to enter and leave chat rooms to converse with others co-located in the same logical room. The chat application is implemented as Java applets run through a Web browser. Figure 7 depicts our implementation of the application. Individual chat rooms are modeled as files exported through WebFS [Vahdat et al. 1996], a file system allowing global URL read/write access. WebFS provides for negotiation of various cache consistency protocols on file open.

We extended WebFS to implement a scalable caching policy suitable to the chat application. In this model, when a user wishes to enter a chat room, the client simply opens a well-known file associated with the room. This operation registers the client with WebFS. Read and write operations on the file correspond to receiving messages from other chatters and sending a message out to the room, respectively. On receiving a file update (new message), WebFS sends the update to all clients which had opened the file for reading (i.e., all chatters in a room). In this case, the client interface applet consists of two threads, a read thread continuously polling the chat file and an event thread writing user input to the chat file. These read/write requests are sent to the chat

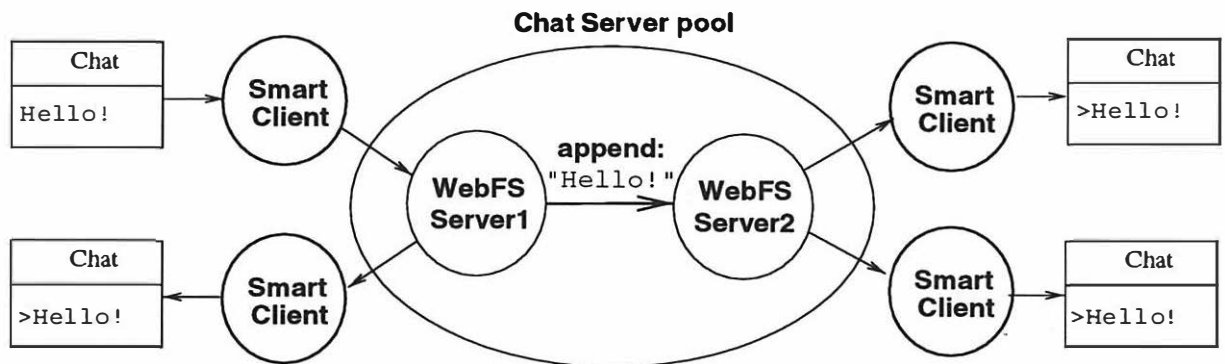
server via the director applet.

The director sends the request to the hostname that represents the best service node at the time. If the request does not complete, the request raises an exception to the director applet. The director applet then calls the service-specific cleanup routine for the request, and resends it to another service node. Note that the request takes a service specific failure event, such as chat file not found or WebFS server is down, and translates it into a general exception. Thus, the director applet can be written for a cluster of machines and reused for many different protocols: FTP, Telnet and chat.

From the above discussion, it is clear that a single WebFS server can quickly become a performance bottleneck as the number of simultaneous users is scaled. To provide system scalability, we allow multiple WebFS servers to handle client requests for a single file. Each server keeps a local copy of the chat file. Upon receiving a client update, WebFS distributes the updates to each of the chat clients connected to it. WebFS also accumulates updates, and every 300 ms propagates the updates to other servers in the WebFS group. This caching model allows for out of order message delivery, but we deemed such semantics to be acceptable for a chat application. If it is determined that such semantics are insufficient, well-known dis-

```
write(http://server1/chat, "Hello!");
```

```
read(http://server2/chat, &x);
```



```
read(http://server1/chat, &x);
```

```
read(http://server2/chat, &x);
```

Figure 7: Implementation of the chat application. Chat rooms are modeled as files with reads corresponding to receiving conversation updates and writes to sending out a message. On a write, the WebFS updates all its clients; the updates are propagated to other servers in a lazy fashion.

tribution techniques [Ladin et al. 1992, Birman 1993] can be used to provide strong ordering of updates.

Since the read requests are idempotent, and the write requests are atomic with respect to WebFS, the chat application is completely tolerant to server crashes. This fault transparency provides the illusion of a single, highly-available chat server machine to the programmer of the Chat client interface applet. Figure 8 demonstrates the behavior of the chat application in the face of a failure to the client's primary server. The graph plots response time as a function of elapsed time. The graph shows that chat delivers less than 5 ms latency to the end user. On detecting a failure, the latency jumps to 1 second before switching to a secondary WebFS server, at which point the latency returns to normal.

## 5 Summary

We have described a solution to the problem of scalability and high-availability which logically migrates server functionality into the client. We will now revisit the goals set forth in Section 1 and examine how Smart Clients addresses each goal:

- **Incremental Scalability** - When a machine is added to or removed from a service group, the director applet supplied by the service updates its list of peer servers. The director applet discovers such modifications through a service-specific mechanism, e.g. keep-alive messages, connecting to a well-known port, or refetching the ser-

vice certificate.

- **Load Balancing** - The director applet maintains a service-specific notion of load (such as number of processes, number of open connections, available bandwidth). Using this information, client requests are routed to the best candidate node.
- **Fault Transparency** - When a failure occurs, the director applet allows the client to clean up any stale state before resending the request to another server. Providing fault transparency requires service support when the request is non-idempotent. For example, in the chat application, the chat service provides atomic writes to the chat transcript.
- **Wide Area Service Topology** - Smart Clients does not place any restriction on topology of server machines. In fact, the director applet can choose an arbitrary site based on considerations such as proximity or predicted performance.
- **Scalable Services To Legacy Servers** - Existing servers that replicate a read-only database can be grouped together for access by Smart Clients. With knowledge of the group set, the director applet can load balance client requests among existing unmodified servers.

Finally, we believe that the architecture presented in this paper can simplify implementation of scalable services with respect to at least fault transparency and load balancing. The Smart Client director provides the illusion of a single, highly available server. This

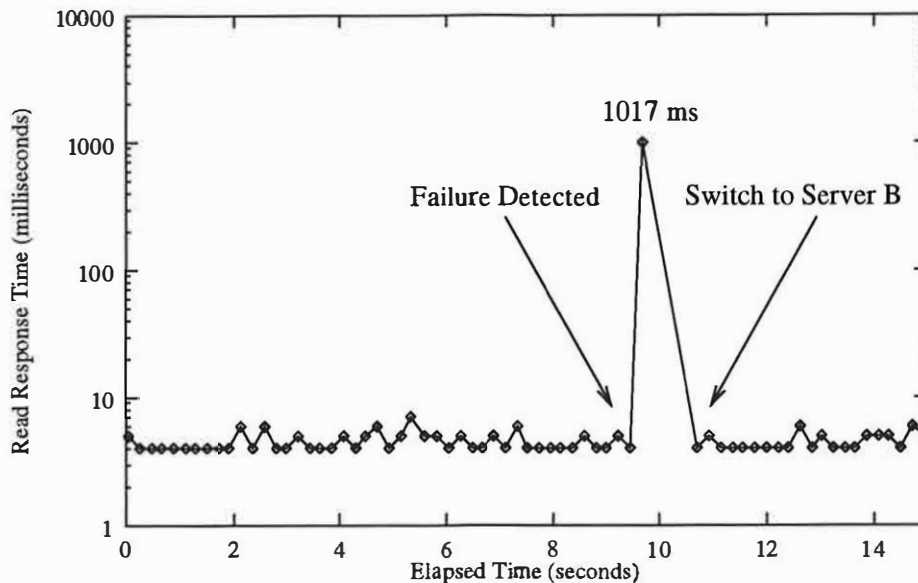


Figure 8: Chat response times in the face of server load. The chat application delivers latencies of approximately 10 ms under normal circumstances. On server failure, the applications takes one second to switch to a peer server.

model substantially decreases the complexity of the client interface applet since this applet need not be concerned with maintaining the set of server nodes. In addition, because of the public interface between the client interface and director applets, each can be written once and interchanged for a number of different services.

## 6 Related Work

The problem of transparently providing fault transparency and load balancing to network services has been addressed previously in a number of contexts. File systems have used server-side replication of volumes and servers to provide fault transparency in systems such as Deceit [Marzullo et al. 1990], AFS [Howard et al. 1988], and HA-NFS [Bhide et al. 1991]. More recently, systems such as xFS [Anderson et al. 1995b] and Petal [Lee & Thekkath 1996] use client-side techniques to improve overall file system performance. Many distributed clusters perform load balancing on the level of jobs (interactive or otherwise) submitted to the system [Nichols 1987, Bricker et al. 1991, Douglass & Ousterhout 1991, Zhou et al. 1992]. Once again, all these systems implement server-side solutions for load balancing and require client intervention to spread jobs among cluster machines.

Perhaps most closely related to Smart Clients are Transaction Processing monitors [Gray & Reuter

1993] (TP monitors). TP monitors provide functionality similar to Smart Clients for access to databases. The TP monitor functions as the director for transactions to *resource managers*, accounting for load on machines, the RPC program number, and any affinity between client and server. Resource managers are usually SQL databases, but can be any server that supports transactions. TP monitors differ from Smart Clients in that they deal exclusively with transactional RPCs as the communication mechanism to the servers. TP monitors are also more closely coupled with the server nodes since they are responsible for starting new server processes.

The Smart Client director can be tailored to each service, while the TP monitor is more of a general purpose director. Smart Clients also provide a bootstrapping mechanism to remove the single point of failure associated with downloading the necessary routing software. In addition, the Smart Client code is significantly more lightweight than the TP monitor which often includes many of the features of traditional operating systems: process management/creation, authentication, and linking resource manager object code with the Transaction Processing operating system (TPOS). This lightweight nature enables Smart Clients to be downloaded into existing Web browsers to customize existing Internet services.

Also related to our systems are ISIS [Birman 1993] and gossip architectures [Ladin et al. 1992] which provide a substrate for developing distributed applica-

tions. ISIS provides reliable group communication to support many of the applications we envision. Gossip architectures use front-ends analogous to Smart Clients to access replicated servers which are kept consistent through lazy updates. Both systems are orthogonal to our work in many respects and still use server-side techniques for much of their functionality.

## 7 Conclusions

In this paper, we have shown that existing solutions to providing transparent access to network services suffer from a lack of knowledge about the semantics of individual services. The recent advent of Java allowing distribution of portable client code presents an opportunity to migrate certain service functionality onto the client machine. We show that such migration can simplify service implementation and improve the quality of service to users. To this end, we describe our implementation of Smart Clients to show the greater flexibility available from a client-side approach to building scalable services. The Smart Clients API provides a generic interface for accessing network services. Further, the decomposition of the API into individual client interface and director applets allows interchanging of these applets for a variety of services. The Smart Clients API is specialized to provide scalable access to three sample services: telnet, FTP, and Internet chat.

In the future, we will further explore service-specific load balancing techniques for achieving scalability. We also plan to demonstrate how Smart Clients can be used to provide load balancing and fault transparency for services replicated across the wide area. We also plan to implement an interface for transparent access to HTTP servers and a fault-tolerant telnet client. Migration of other code, besides the director, from the server to the client will also be explored.

## Acknowledgments

We would like to thank our shepherd Rob Gingell, Bruce Mah, Rich Martin, and Neal Cardwell for their help in substantially improving the presentation of this paper. We would also like to thank Thomas Wendt for providing the source to jfox, the Java Web browser we modified to support Smart Clients.

## References

- [Anderson et al. 1995a] T. E. Anderson, D. E. Culler, D. A. Patterson, and the NOW Team. "A Case for NOW (Networks of Workstations)". *IEEE Micro*, February 1995.
- [Anderson et al. 1995b] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. "Serverless Network File Systems". In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pp. 109–126, December 1995.
- [Anderson et al. 1996] E. Anderson, D. Patterson, and E. Brewer. "The Magicrouter, an Application of Fast Packet Interposing". May 1996. Submitted For Publication. Also see <http://HTTP.CS.Berkeley.EDU/~eanders-/magicrouter/>.
- [Berners-Lee 1995] T. Berners-Lee. "Hypertext Transfer Protocol HTTP/1.0", October 1995. HTTP Working Group Internet Draft.
- [Bhide et al. 1991] A. Bhide, E. N. Elnozahy, and S. P. Morgan. "A Highly Available Network File Server". In *Proceedings of the 1991 USENIX Winter Conference*, pp. 199–205, 1991.
- [Birman 1993] K. P. Birman. "The Proecss Group Approach to Reliable Distributed Computing". *Communications of the ACM*, 36(12):36–53, 1993.
- [Bricker et al. 1991] A. Bricker, M. Litzkow, and M. Livny. "Condor Technical Summary". Technical Report 1069, University of Wisconsin—Madison, Computer Science Department, October 1991.
- [Brisco 1995] T. Brisco. "DNS Support for Load Balancing", April 1995. Network Working Group RFC 1794.
- [Dig 1995] Digital Equipment Corporation. *Alta Vista*, 1995. <http://www.altavista.digital.com/>.
- [Douglass & Ousterhout 1991] F. Douglass and J. Ousterhout. "Transparent Process Migration: Design Alternatives and the Sprite Implementation". *Software - Practice and Experience*, 21(8):757–85, August 1991.
- [Ghormley et al. 1995] D. Ghormley, A. Vahdat, and T. Anderson. "GLUnix: A Global Layer UNIX for NOW". See <http://now.cs.berkeley.edu/GLunix/glunix.html>, 1995.
- [Goldstein & Dale 1995] I. Goldstein and P. Dale. "A Scalable, Fault Resilient Server for the WWW". OSF ARPA Project Proposal, 1995.
- [Gosling & McGilton 1995] J. Gosling and H. McGilton. "The Java(tm) Language Environment: A White Paper". <http://java.dimensionx.com/whitePaper/java-whitepaper-1.html>, 1995.
- [Gray & Reuter 1993] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[Anderson et al. 1995a] T. E. Anderson, D. E. Culler, D. A.

- [Howard et al. 1988] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. "Scale and Performance in a Distributed File System". *ACM Transactions on Computer Systems*, 6(1):51-82, February 1988.
- [Jav 1996] JavaSoft. *Java RMI Specification, Revision 1.1*, 1996. See <http://chatsubo.javasoft.com/current/doc/rmi-spec/rmiTOC.doc.html>.
- [Katz et al. 1994] E. D. Katz, M. Butler, and R. McGrath. "A Scalable HTTP Server: The NCSA Prototype". In *First International Conference on the World-Wide Web*, April 1994.
- [Ladin et al. 1992] R. Ladin, B. Liskov, L. Shirira, and S. Ghemawat. "Providing Availability Using Lazy Replication". *ACM Transactions on Computer Systems*, 10(4):360-391, 1992.
- [Leach 1996] P. Leach. Personal Communication, November 1996.
- [Lee & Thekkath 1996] E. K. Lee and C. A. Thekkath. "Petal: Distributed Virtual Disks". In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.
- [Marzullo et al. 1990] K. Marzullo, K. Birman, and A. Siegel. "Deceit: A Flexible Distributed File System". In *Proceedings of the 1990 USENIX Summer Conference*, pp. 51-61, 1990.
- [Mat 1996] Matrix Information and Directory Services, Inc. *MIDS Internet Weather Report*, 1996. See <http://www2.mids.org/weather/index.html>.
- [Net 1994] Netscape Communications Corporation. *Netscape Navigator*, 1994. <http://www.netscape.com>.
- [Nichols 1987] D. Nichols. "Using Idle Workstations in a Shared Computing Environment". In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pp. 5-12, November 1987.
- [Vahdat et al. 1996] A. Vahdat, M. Dahlin, and T. Anderson. "Turning the Web into a Computer". May 1996. See <http://now.cs.berkeley.edu/WebOS>.
- [Wendt 1996] T. Wendt. *Jfox*, 1996. <http://www.uni-kassel.de/fb16/ipm/mt/java/jfox.html>.
- [Wetherall & Tennenhouse 1995] D. Wetherall and D. L. Tennenhouse. "Active Networks: A New Substrate for Global Applications". 1995. Submitted for Publication.
- [Zhou et al. 1992] S. Zhou, J. Wang, X. Zheng, and P. Delisle. "Utopia: A Load Sharing Facility for Large, Heterogeneous Distributed Computing Systems". Technical Report CSRI-257, University of Toronto, 1992.



# Building Distributed Process Management on an Object-Oriented Framework

Ken Shirriff

*Sun Microsystems*  
2550 Garcia Avenue  
Mountain View, CA 94043  
shirriff@eng.sun.com

## Abstract

The Solaris MC distributed operating system provides a single-system image across a cluster of nodes, including distributed process management. It supports remote signals, waits across nodes, remote execution, and a distributed `/proc` pseudo file system. Process management in Solaris MC is implemented through an object-oriented interface to the process system. This paper has three main goals: it illustrates how an existing UNIX operating system kernel can be extended to provide distributed process support, it provides interfaces that may be useful for general access to the kernel's process activity, and it gives experience with object-oriented programming in a commercial kernel.

## 1 Introduction

The Solaris MC research project<sup>1</sup> has set out to extend the Solaris® operating system to clusters of nodes. We believe that due to technology trends, the preferred architecture for large-scale servers will be a cluster of off-the-shelf multiprocessor nodes connected by an industry-standard interconnect. The Solaris MC operating system is a prototype distributed operating system that provides a *single system image* for such a cluster and will provide high availability so that the cluster can survive node failures. Solaris MC is built as a set of extensions to the base Solaris UNIX® system and provides the same ABI/API as the Solaris OS, running unmodified applications.

As process management is a key component of an operating system, a single-system image cluster operating system must implement a variety of process-related operations. In addition to creating and destroying processes, a POSIX-compliant[13] operating system

must support signalling a process, waiting for child process termination, managing process groups, and handing tty sessions. In addition, Solaris provides a file-based interface to processes through the `/proc` file system, which is used by `ps` and debuggers. Finally, cluster-based process management must provide additional functionality such as controlling remote execution of processes.

Our decision to base the Solaris MC operating system on the existing Solaris kernel influenced many of the architectural decisions in the process management subsystem. One goal was to minimize the amount of kernel change required, while making the changes necessary to provide a fully-transparent single-system image. Thus, the Solaris MC process subsystem can be distinguished on one hand from cluster process systems that run entirely at user level (such as GLUnix [22]) and on the other hand from systems that build distributed process management from scratch (such as Sprite[18]). By putting distributed process support in the kernel, we gain both better performance and more transparency than systems that operate at user level. By using most of an existing kernel, we reduce the development cost and increase the commercial potential. This is similar in concept to Locus [20] and OSF/1 AD TNC [25].

As well as describing an implementation of cluster process management, this paper describes interfaces into the kernel's process management. UNIX systems lack a good kernel interface allowing access to the low-level process management. In comparison, the file system has a well-defined existing interface between the kernel and the underlying storage system through `vnodes` [15], allowing new storage systems or distributed file systems to be "plugged in." The object-oriented interfaces described below are a first cut at providing similar access to processes.

This paper describes the process management subsystem of the Solaris MC operating system. Section 2 provides an overview of the implementation of distrib-

1. Solaris MC is an internal name of a research project at Sun Microsystems Laboratories. More information on the project can be obtained from <http://www.sunlabs.com/research/solaris-mc>

uted process management. Section 3 discusses the interfaces that the Solaris MC process management uses. Section 4 gives implementation details, Section 5 gives performance measurements, Section 6 compares this system with other distributed operating systems, and Section 7 concludes the paper.

## 2 Distributed process management

### 2.1 Overview of the Solaris MC operating system

The Solaris MC operating system [14] is a prototype distributed operating system for a cluster of nodes that provides a single-system image: a cluster appears to the user and applications as a single computer running the Solaris operating system. Solaris MC is built as a set of extensions to the base Solaris UNIX system and provides the same ABI/API as Solaris, running unmodified applications. The Solaris MC OS provides a single-system image through a distributed file system, clusterwide process management, transparent access from any node to one or more external network interfaces, and cluster administration tools. High availability is currently being built into the Solaris MC OS: if a node fails, the remaining nodes will remain operational, and the file system and networking will transparently recover.

The Solaris MC OS is built as a collection of distributed objects in C++, for several reasons. First, the object-oriented approach gives us a good mechanism for ensuring that components communicate through well-defined, versionable interfaces; we use the IDL interface definition language [21]. Second, a distributed object framework lets us invoke methods on objects without worrying if they are local or remote, simplifying design and programming. Finally, the object framework keeps track of object reference counting, even in the event of failures, and provides subsystem failure notification, simplifying the design for high availability.

We use an object communication runtime system [5] based on CORBA [16] that borrows from Solaris doors for interprocess communication and Spring subcontracts [10] for flexible communication semantics. Our object framework lets us define object interfaces using IDL [21]. We implement these objects in C++, and they can reside either at user level or in the kernel. References to these objects can be passed from node to node. A method on an object reference is invoked as if the object were a standard C++ object. If the object is remote, the object framework will transparently marshal arguments, send the request to the object's node, and return the reply. Thus, the object framework provides a convenient mechanism for implementing distributed objects and providing communication.

The Solaris MC proxy file system (PXFS) provides an efficient, distributed file system for the cluster. The file system uses client and server-side caching using the Spring caching architecture, which provides UNIX consistency semantics. The file system is built on top of vnodes, as shown in Figure 1: on the client side, the file system looks like a normal vnode-based file system, and on the server side an existing file system (such as UFS) plugs into the Solaris MC file system using the vnode interface to provide the actual file system storage. The components of the Solaris MC file system communicate using the object system.

### 2.2 Distributed process management

The process management component of Solaris MC was designed to satisfy several goals:

- Provide binary compatibility and POSIX semantics.
- Minimize changes to the kernel.
- Use an object-oriented design based on CORBA and IDL.
- Provide good performance: The system should avoid node-to-node communication for local process operations, for instance.
- Provide reliability in the face of failures: The system should be able to continue working if nodes fail or are added to the system.

To support these goals, we implemented process management using the architecture shown in Figure 2. The process management globalization code is written as a C++ module that is loaded into the kernel address space. Conceptually, this module sits on top of the existing process code; system calls are directed into the

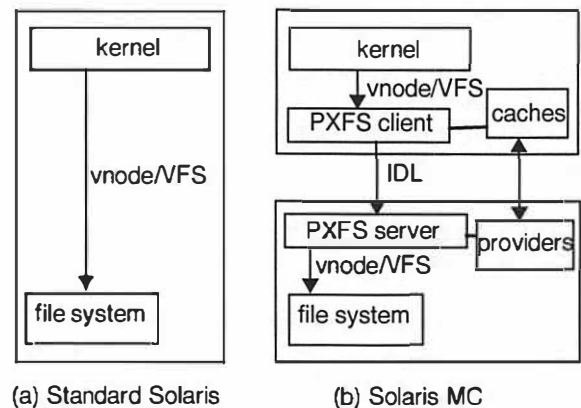


Figure 1: Structure of the Solaris MC proxy file system (PXFS). Solaris MC splits the file system across multiple nodes by adding PXFS client and PXFS server layers above the underlying file system. These layers communicate through IDL-based interfaces, as do the file system caches.

globalization layer. Much of the kernel's existing process management (*e.g.* threads, scheduling, and process creation) were used unchanged, some kernel operations were modified to hook into the globalization layer (*e.g.* fork, exec, and signals), and a few components of the kernel were largely replaced (*e.g.* wait and process groups). The distributed `/proc` file system is implemented independently, as will be discussed in Section 4.6.

We made the design decision to have a virtual process object associated with each process and a process manager object associated with each node. The virtual process object handles process-specific operations and holds the state necessary to globalize the process, while the process manager object handles node-specific operations (such as looking up a process ID or pid) and looks after the node-specific state. These objects communicate using IDL interfaces to perform operations.

A second key design decision was to keep most kernel process fields accurate locally, so most kernel code can run without modification. For instance, the local process ID is the same as the global process ID, rather than implementing a global pid space on top of a different local pid space. One alternative would be to do away with the existing kernel pid data structures and only use the globalized data. This approach would require extensive changes to the kernel, wherever these data structures were accessed. In our approach, the kernel needed to be modified only when the local and global pictures wouldn't correspond, for instance when children are accessed or a process is accessed by pid. Another approach would be to use separate local and global pids; this would require mapping between them whenever the user supplies or receives a pid. This would reduce the kernel changes, but the mapping step would

make all operations on pids slower and more complicated.

The main objects used in the system are shown in Figure 3. The standard Solaris OS stores process state in a structure called `proc_t`. We added a field to this structure to point to the virtual process object; since existing Solaris code passes around `proc_t` pointers, we need some way to get from this to the virtual process object. The virtual process object manages the global parent/child relationships through object references to the parent and children (which may be on the same or different nodes). It also contains global process state information, exit status for children, and forwarding information for remotely execed processes. Additional objects (discussed in Section 4.2) manage process groups and sessions.

The process manager contains the node-specific data. In particular, it holds a map from pids to virtual process objects for all local processes and processes originally created on this node. (This is analogous to the home structure in MOSIX [3].) It also contains object references to the process managers on the other nodes in the cluster. Thus, the process manager is used to locate processes, to locate other nodes, and to perform operations that aren't associated with a particular process (*e.g.* `sigsendset`).

To locate processes, we partitioned the pid space among the nodes, so the top digits of the pid hold the node number of the original node of the process (similar to OSF/1 [25] or Sprite [18]). Note that a process may move from node to node through the remote `exec` operation; since the pid can't change, the pid will specify the original node, not necessarily the current node. Thus, given a pid it is straightforward to determine the

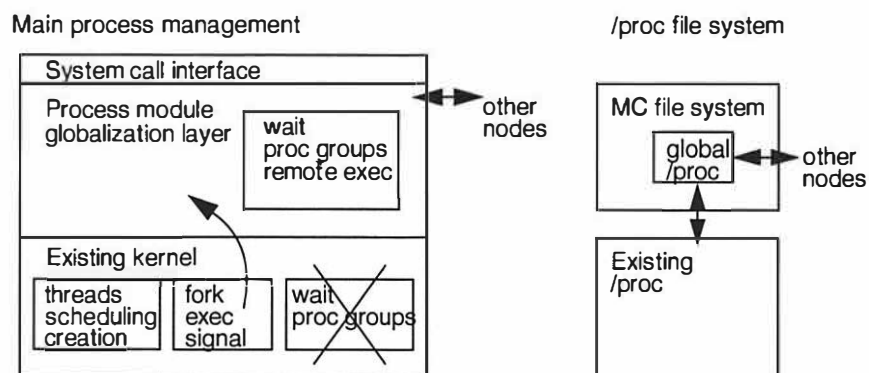


Figure 2: Structure of process management in the Solaris MC operating system. Process management is divided into two independent components. The main component, which supports UNIX process operations, is implemented as a module that interacts with the kernel to provide globalized process operations. The `/proc` component, which provides a global implementation of the `/proc` file system, is implemented as part of the Solaris MC PXFS file system.

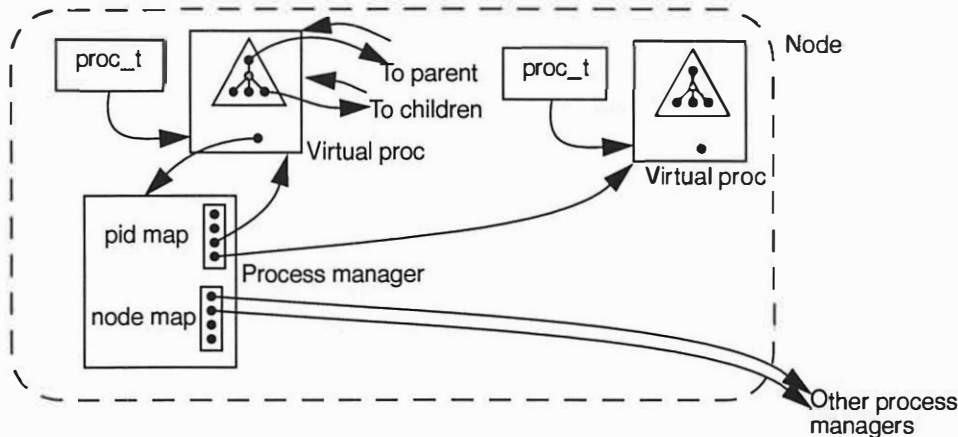


Figure 3: The objects for process management. For each local process, the existing Solaris `proc_t` structure points to a virtual process structure that holds the parent and child structure as well as other globalizing information. The process manager has a map from pids to the virtual processes and from the node ids to the process managers. Solid circles indicate object references.

original node. The process manager on this node can be queried for an object reference to the virtual process object, and then the operation can be directed to that object. Instead of updating all references when a process changes nodes, we leave a forwarding reference in a virtual process object on the original node. When this is accessed, it sends a message to the requestor and the stale pointer is updated. Because the object framework does reference counting, the forwarding object will automatically be eliminated when there are no longer any references to it.

The virtual process object in Solaris MC can be compared with the global process structures in OSF/1 AD TNC [25]. In the OSF/1 system, the only process structure visible to the base server is the simple `vproc` structure, which is analogous to a `vnode` structure. The `vproc` structure contains the `pid`, an operation vector, and a pointer to internal data. This pointer references a `pvproc` structure, which holds the globalization information (location, parent-sibling-child relationships, group, and session), and another operation vector which directs operations to local process code or a remote operation stub that performs a RPC.

In both systems, the virtual process layer sits between the application's process requests and the underlying physical process code, providing globalization. An OSF/1 `vproc` structure is roughly analogous to a Solaris MC object reference, while the `pvproc` holds the actual data and roughly corresponds to the Solaris MC virtual process object. Thus, OSF/1 encapsulates process data by placing it in a separate `pvproc` structure, while Solaris MC encapsulates it inside an object. One key difference occurs on nodes that access a process that is located on a different node. In OSF/1, these nodes will

have both a `vproc` and `pvproc` structure and the associated process state, while Solaris MC will only have an object reference. A second difference is that OSF/1 directs operations remotely or locally through an operation vector in the `pvproc` structure, while Solaris MC directs operations remotely or locally through the method table associated with the virtual process object reference, which is managed automatically by the object framework. Thus, the object framework hides the real location of a Solaris MC virtual process object. Finally, the marshalling and transport of requests between nodes is performed transparently by the Solaris MC object subsystem, while in OSF/1, the `vproc` system needed its own communication system, using the TNC message layer.

### 3 Process interfaces

Unlike the file system with its `vnode` interface, the process control in Solaris lacks a clean interface for extension. Adding a file system to UNIX used to be a difficult task. Early attempts at distributed file systems (e.g. Research Version 8 [1] and EFS [6]) used *ad hoc* interfaces, while NFS [26] and RFS [2] used more formal interfaces, `vnodes` and the file system switch respectively. These interfaces required extensive restructuring of the kernel, but the result is that adding a new file system is now relatively straightforward.

The same restructuring and kernel interface needs to be considered for the process layer in the kernel. The `/proc` file system provides one interface to manipulate processes, but it is limited in functionality to providing status and debugging control. The Locus `vproc` architecture [25] and the Solaris MC virtual process object

can be considered steps towards a general interface analogous to `vnodes`.

One approach would be to provide distributed process support through extensions to the `/proc` interface; we didn't take this approach for several reasons. First, a file system based interface has a serious "impedance mismatch" with the operations that take place on processes; a simple process system call would get mapped to an open-write-close. Second, the overhead of opening files to manipulate processes is undesirable. Finally, a `/proc`-based interface would probably result in an assortment of `ioctl`s more confusing than the current `/proc` `ioctl`s; we desired a cleaner interface.

We broke the process interface into two parts: a procedural and an internal object-oriented interface. The procedural interface is used for communication between the existing kernel and the globalization layer; this interface is described in Section 3.1. The object-oriented interface is used by the global process implementation, and is described in Section 3.2.

### 3.1 Procedural interface

The procedural interface, given in Table 1, is a thin layer on top of the object-oriented interface for use by the existing C kernel, for both the system call layer and the underlying kernel process code to communicate with the globalization layer.

Most of the procedural calls are used to direct system calls into the globalization layer: `VP_FORK`, `VP_EXEC`, `VP_VHANGUP`, `VP_WAIT`, `VP_SIGSENDSET`, `VP_SETSID`, `VP_PRIOCNLSET`, `VP_GETSID`, `VP_SIGNAL`, and `VP_SETPGID`. Note that operations such as `getpid`, `nice`, or `setuid` don't go into the globalization layer as they operate on the current (local) process.

The up-calls from the kernel to the globalization layer are `VP_FORKDONE`, at the end of a fork to allow the globalization layer to update its state; `VP_EXIT`, when a process exits, `VP_SIGCLDMODE`, when a process changes

its `SIGCLD` handling flags; and `VP_SIGCLD`, when a process would send a `SIGCLD` to its parent.

These procedural calls are implemented by having `VP_FORK`, `VP_EXIT`, `VP_SETSID`, `VP_VHANGUP`, `VP_WAIT`, `VP_SIGCLD`, and `VP_SIGCLDMODE` call the virtual process object associated with the current process, while the remaining operations call the node object on the current node. This is an optimization; since the former set of operations act on the current process it is more efficient to send the operations to the virtual process object directly.

### 3.2 Internal interface

The object-oriented interface to the node object is given in Table 2. The node methods consist of operations to manage node state (`addvpid`, `registerloadmgr`, `getprocmgr`, `addprocmgr`, `findvproc`, `findvpid`, `lockpid`, `unlockpid`), operations that create new processes (`rfork`, `rexec`), and operations that act on multiple processes (`sigsendset`, `priocntlset`).

The interface to the process object is given in Table 3 and contains methods that act on an existing process. Most of these methods correspond to process system calls. The `childstatuschange` and `releasechild` methods are used for waits; they are described in more detail in Section 4.4. The `childmigrated` method keeps the parent informed of the child's location.

One potential future path is to merge the `/proc` file system with the object-oriented interface, allowing `/proc` operations to be performed through the object interface. The `/proc` file system would then be just a thin layer allowing access to the objects through the file system. This would make the process object into a single access path into the process system. It would also simplify the kernel code by combining all the process functionality into one place. Now, the Solaris `/proc` code is entirely separate from the virtual process code.

|   |   |
|---|---|
| <code>VP_FORK(flag, pid, node)</code>         | <code>VP_SIGSENDSET(psp, sig, local)</code>                     |
| <code>VP_EXIT(flag, status)</code>            | <code>VP_SETSID(flag, rval)</code>                              |
| <code>VP_EXEC(fname, argp, envp, node)</code> | <code>VP_PRIOCNLSET(version, psp, cmd, arg, rval, local)</code> |
| <code>VP_VHANGUP()</code>                     | <code>VP_GETSID(pid, pgid, sid)</code>                          |
| <code>VP_WAIT(idtype, id, ip, options)</code> | <code>VP_SIGNAL(pid, sigsend)</code>                            |
| <code>VP_SIGCLD(cp)</code>                    | <code>VP_SETPGID(pid, pgid)</code>                              |
| <code>VP_SIGCLDMODE(mode)</code>              | <code>VP_FORKDONE(p, cp, mig)</code>                            |

Table 1: Procedural calls into the virtual process layer. These calls are just a thin layer between the C code of the Solaris kernel and the C++ code of the Solaris MC module

## 4 Details of process management

This section describes the main process functions in more detail. Sections 4.1, 4.2, 4.3, and 4.4 describe how we implemented signals, process groups, remote execution, and cross-node waits, respectively. Section 4.5 discusses what failure recovery will be provided. Section 4.6 explains how a global `/proc` file system was added to the Solaris MC file system. Finally, Section 4.7 discusses our experience with object-oriented programming in the kernel.

### 4.1 Signals

Several issues complicate signal delivery in a distributed system. Delivery of a signal to a remote process is straightforward; an object reference to the process is obtained from the `pid`, the signal method is invoked on that process object, and the object delivers the signal locally.

The more complicated cases are a kill to a process group, the `sigsend` system call, and the `sigsendset` system call. To signal a process group, the appropriate process group object is located, the signal method is invoked on this object, the process group object invokes the member process objects, and these object deliver the signals. Locking complicates this operation.

The `sigsend` and `sigsendset` system calls allow signals to be sent to complex sets of processes, formed by selecting sets of processes based on `pid`, group id, session id, user id, or scheduler class, and then combining these sets with boolean operations. To handle these system calls, the operation is sent to each node in the system and each node signals the local processes

matching the set. This can be inefficient since all nodes must handle the operation; if performance becomes a problem, we will add optimizations for the simple cases (such as the set specifies a single process or process group) so they can be handled efficiently.

### 4.2 Process groups

POSIX process groups and sessions support job control: a `tty` or `pseudo-tty` corresponds to a session, and each job corresponds to a process group. The process management subsystem must keep track of which processes belong to which process groups, and which process groups belong to which sessions. Process group membership is used by the I/O subsystem to direct I/O only to processes in a foreground process group, and send `TTIN`/`TTOUT` signals to background processes.

A POSIX-compliant system must also detect orphaned process groups (which are unrelated to orphaned processes). A process group is not orphaned as long as there is a process in the group with a parent outside the process group but inside the session. (The two prototypical orphaned process groups are the shell and a job that lost its controlling shell.) Orphaned process groups are not permitted to receive terminal-related stop signals, as there is no controlling process to return them to the foreground. In addition, if a process group becomes orphaned and contains a stopped process, the process group must receive `SIGHUP` and `SIGCONT` signals to prevent the processes from remaining stopped forever.

To support these POSIX semantics, the Solaris MC OS uses an object for each process group and an object for each session, with the session object referencing the

|  |   |
|--|---|
| <code>addvpid(pid, vproc, update)</code>                           | <code>rfork(state, astate, flags)</code>                      |
| <code>registerloadmgr(mgr)</code>                                  | <code>rexec(fname, argp, envp, state, flags, newvproc)</code> |
| <code>sigsendset(psp, sig, local, srcpid, cred, srcsession)</code> | <code>lockpid(pid)</code>                                     |
| <code>priocntlset(version, psp, cmd, arg, rvp, local)</code>       | <code>unlockpid(pid)</code>                                   |
| <code>getprocmgr(nodenum)</code>                                   | <code>findvproc(pid, local, vproc)</code>                     |
| <code>addprocmgr(nodenum, pm)</code>                               | <code>freevpid(pid)</code>                                    |

Table 2: Methods on the node object

|   |  |
|---|--|
| <code>signal(srcpid, cred, srcsession, signal)</code> | <code>childmigrated(childpid, newchild)</code>   |
| <code>setpgid(srcpid, pgid)</code>                    | <code>parentgroupchange(par_pgid, par_sid)</code>                                      |
| <code>getsid(srcpid, pgid, sid)</code>                | <code>childstatuschange(childpid, wcode, wdata, utime, stime, zombie, noparent)</code> |
| <code>getpid()</code>                                 | <code>setsid(sess)</code>  |
| <code>releasechild(mode, noparent)</code>             |  |

Table 3: Methods on the virtual process object

member process group objects and each process group object referencing the member processes. For efficiency, we originally planned to include an additional object, the local process group object, which would reside on each node that had a process in the process group and would reference the local processes in the process group. The local process group object would be more efficient for signalling process groups split across multiple nodes and for determining orphaned process groups, since most messages would be intra-node, to the local process group object, rather than inter-node, to the top-level process group object. The added complexity of this approach, however, convinced us to eliminate the local process group objects.

The process group objects detect orphaned process groups by keeping a reference count of the number of controlling links to the process group (*i.e.* processes with a parent outside the process group but inside the session). Operations that create or destroy a link (changing process group membership or exiting of the parent or child) inform the process group object; if the count drops to zero, the process group is signalled as required.

### 4.3 Remote execution

The Solaris MC operating system provides a remote execution mechanism to create processes on remote nodes. The `rexec` system call is used for remote execution; it is similar to the UNIX `exec` system call except the call specifies a node where the new process image runs. The remote process then sees the same environment as if it were running locally, due to the global file system and other single-system-image features.

Process migration, which moves an existing process to a new node, is not yet implemented in Solaris MC, due to the additional complexity of moving the address space. The remote `exec` is much easier to implement and more efficient than migrating the address space of a running process (which could be partially paged out). We feel that most of the load-balancing benefit of remote processes can be obtained by positioning the process at execution time (as do the Plan 9 researchers [19], although Harchol-Balter and Downey [11] have found benefits from moving running processes). We plan to implement migration of running processes later, in particular to off-load processes from a node scheduled for maintenance; at this time we will provide `rfork` and migration (which are very similar, since a `rfork` is basically a `fork` and a `migrate`). The implementation of migration in Solaris MC will be similar to that in other distributed operating systems (*e.g.* Sprite [7], MOSIX [3], or Locus [20]), moving the address space to the new node.

The `rexec` call operates by packaging up the necessary state of the process (the list of open files, the `proc_t` data, and the list of children), and sending this to the destination node along with the `exec` arguments. The process is then started on the remote node (using a hook into low-level process creation, since the existing `fork` code won't work without a local parent) and the old process is eliminated. Finally, the parent receives an object reference to the new virtual process object to update its child list. Our process management code then transparently manages the cross-node parent-child relationships, as described in Section 4.4.

Remote execution takes advantage of the Solaris MC distributed file system to handle migration of open files across remote `execs`. With the distributed file system, it doesn't matter what node is performing a file operation. For each open file, the offset and the reference to the file object are sent across to the new node. These are then used to create `vnodes` for the new process. File operations from the new process then operate transparently, with the Solaris MC file system ensuring cache consistency.

The Solaris MC operating system provides support for multiple load balancing policies. A node can be specified in the `rexec` call, or the location can be left up to the system. By default, `rexec` uses round-robin placement if a node isn't specified, but hooks are provided for an arbitrary policy; the `registerloadmgr` method allows a load management object to be registered with the node manager. For each remote execution, the node manager will then query the load management object for the destination node id, which can be generated by any desired algorithm. The load manager can use algorithms similar to the OSF/1 AD TNC [25] or Sprite [7] load daemons. Since the load management object communicates with the node manager through the Solaris MC object model, the load management object can be implemented either in the kernel or at user level, even on a different node. This illustrates the flexibility and power of our distributed object model.

### 4.4 Waits

The Solaris MC operating system supports the UNIX wait semantics, even when the parent and child are on different nodes.

Several approaches are possible for handling waits. One model, the "pull" model, has the parent request information from the child when it does a wait. If the wait does not return immediately, the parent sets up callbacks with the children so that it is informed when an event of interest occurs. A second model, the "push"

model, has the children inform the parent of every state change.

The pull model has the advantage that no messages are exchanged until the parent actually performs a wait. However, setting up and tearing down the callbacks can be expensive if the parent has many children, since many callbacks will have to be created and removed for each wait.

The Solaris MC operating system uses the push model. If child processes perform many state changes that the parent doesn't care about, this will be more expensive. Normally, however, the process will just exit, requiring one message. It would be interesting to compare the number of messages required for the push vs. pull models on a real workload, however.

Note that in Solaris an exiting child process normally goes into a "zombie" state until the parent performs a wait on it. It would have been simpler to do away with zombie processes altogether and just keep the child's exit status with the parent, but Solaris semantics require a zombie process which will show up on `ps`, for instance. In addition, POSIX requires that the `pid` isn't reused until the wait is done.

The Solaris MC operating system uses a simple state machine to handle waits and avoid race conditions, as shown in Figure 4. When the child exits, it informs the parent (through the `childstatuschange` method) and changes state to the right; when the parent exits or waits on the exited child, it informs the child (through the `releasechild` method) and the child changes state downwards. When the child reaches the final state, it can be freed from the system.

Thus, when a child exits, the parent's virtual process object keeps track of the state of the child (*i.e.* exited), its `exit` status, and the user and system time used by the child process (for use by `times(2)`).

#### 4.5 Failure recovery

The Solaris MC operating system is designed to keep running in the event of node failures. The semantics for a node failure are that the processes on the failed node

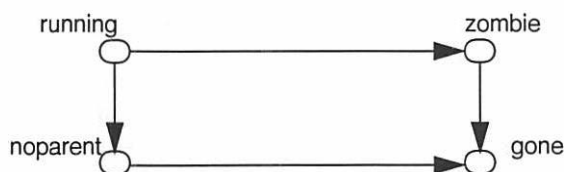


Figure 4: The state machine for processes. If a process exits, its state transitions to the right. If its parent exits, its state transitions downwards.

will die, and to the rest of the system it should look as if these processes were killed. That is, the normal semantics for waits should apply, ensuring that zombie processes don't result.

In the event of a failure, the system must recover necessary state information that was on the failed node. One type of information is migration pointers. If a process did multiple `rexecs`, it may have forwarding pointers on the failed node. After failure, it must be linked up with the parent. Also, after failure, process group membership must be checked to see if an orphaned process group resulted.

To support failure recovery, process management in the Solaris MC OS was designed to avoid single points of failure. For instance, there is no global `pid` server (this also helps efficiency). A remotely `exec'd` process can lose its home node through a failure; in this case, the next live node in sequence will take over as home node, providing a way to find the process.

Failure recovery in the Solaris MC operating system is still being developed and will be based on low-level failure handling in the object layer. The object layer will notify clients of failed invocations, will clean up references to the failed node, and will release objects that are no longer referenced from a live node.

#### 4.6 The `/proc` file system

The `/proc` file system is a pseudo file system in Solaris [9] that provides access to the process state and image of each process running in the system. This file system is used principally by `ps` to print the state of the system and by debuggers such as `dbx` to manipulate the state of a process. Each process has an entry in `/proc` under its process id, so the process with `pid nnnnn` is accessible through `/proc/nnnnn`. The `/proc` file system supports reads and writes, which access the underlying memory of the process, and `ioctl`s, which can perform arbitrary operations on a process such as single stepping, returning state, or signalling. By providing a global `/proc`, the Solaris MC OS supports systemwide `ps` and cross-node debugging.

The `/proc` file system raises several implementation issues for a distributed process system. First, all processes on the system must appear in the `/proc` directory. Second, operations on a `/proc` entry must control the process wherever it resides. Third, `ioctl`s may require copying arbitrary amounts of data between user space on one node and the kernel on another. Finally, some `/proc` `ioctl`s return file descriptors for files that the process has opened; these file descriptors must be made meaningful on the destination node.

In Solaris MC, `/proc` is extended to provide a view of the processes running on the entire cluster, by merging together local `/procs` into a distributed picture, as shown in Figure 5. Thus, each node uses the existing `/proc` implementation to provide the actual `/proc` operations and a merge layer makes these look like a global `/proc`. To implement this, the `readdir` operation was modified to return the contents of all the local `/procs`, merged into a single directory. The new pathname lookup operation returns the `vnode` entry in the appropriate local `/proc`, so operations on the `/proc` entry then automatically go to the right node. If the process migrates away, the file system operation generates a “migrated” exception internally; the merging layer catches this and transparently redirects the operations.

The implementation of the distributed `/proc` file system illustrates the advantages of an object-oriented file system implementation. The merging `/proc` file system was implemented as a subclass of the standard Solaris MC distributed file system that redefines the `readdir` and `lookup` operations. The remainder of the file system code is inherited unmodified. Thus, the object-oriented approach allows new file system semantics to be implemented while sharing most of the old code.

The `/proc` file system uses two techniques for handling cross-node `ioctl`s. One solution would be to modify the `/proc` implementation code to explicitly copy the argument data between the two nodes. This would, however, require modifications to the `/proc` source. The solution used in Solaris MC extends the technique used by the Solaris MC file system; the low-level routines to copy data to and from the kernel (`copyin`, `copyout`, `copyinstr`, `copyoutstr`) were modified so that if they are called in the course of an `ioctl`, the data is transferred from the remote node. A few `ioctl`s, however, required special handling because

they return the numeric value of a file descriptor corresponding to an open file. For these `ioctl`s, the open file is wrapped in a Solaris MC PXFS file system object and the object reference is transferred to the client node. A new file descriptor is opened on the client corresponding to this object. Then, any operations on this file descriptor will be sent through PXFS back to the original open file.

## 4.7 Experience with objects

Our experience with object-oriented programming in the kernel is generally positive. Objects provided a cleaner design because of the encapsulation of data structures and the enforcement of well-defined interfaces. The object framework simplified implementation because of the location-independence of object invocations. This obviates having to keep track of which processes are local and which processes are global. It also makes remote procedure calls entirely transparent.

One major difficulty we found with object-oriented programming for the kernel is that the C++ tools aren’t as well developed as for C. We encountered several compiler and linker difficulties, and templates remain a problem. In addition, our C++ debugging environment is rather primitive. For instance, we had to modify `kadb` to return demangled C++ names.

A second issue with object-oriented programming is efficiency; our problems generally arose from over-enthusiastic use of C++ features. One problem the Solaris MC project encountered was that C++ exceptions were costly in our implementation, both when they were thrown and even when they weren’t (since the additional code polluted the instruction cache). As a result, we decided to remove C++ exceptions from the Solaris MC implementation and use a simple error return mechanism instead. A second problem is that object-oriented programming makes it easy to have an excessive number of classes implementing nested data

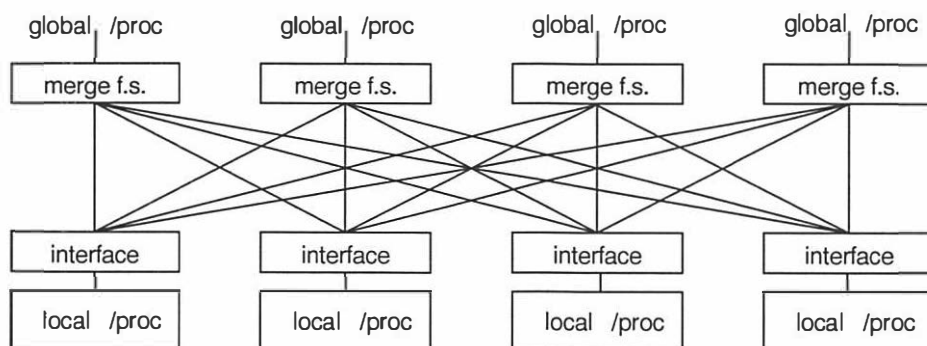


Figure 5: The distributed `/proc` file system merges together the local `/proc` file systems so each node has a `/proc` that provides a global view of the system.

abstractions and multiple levels of subclasses to provide specialization. Unfortunately, this can result in a very deep call graph, with the associated performance penalty. We are currently restructuring the implementation of our transport layer, where performance suffers due to this problem.

## 5 Performance measurements

Tables 4 and 5 give performance measurements of the Solaris MC process management implementation.<sup>2</sup> These measurements were taken on a cluster of four two-processor SPARCstation 10's, three with 50 MHz processors and one with 40 MHz processors, each with 64 MB and running Solaris 2.6 with Solaris MC modifications. The interconnect is a 10 MByte/sec 100-Base T using the SunFast interface, and the transport layer uses the STREAMS stack. Note that Solaris MC is a research prototype and has not been tuned for performance. Thus, these numbers should be viewed as very preliminary and not a measure of the full potential of the system.

### 5.1 Micro-benchmarks

Table 4 gives several performance measurements comparing the unmodified Solaris OS, the Solaris MC OS performing local operations, and the Solaris MC OS performing remote operations.

The first three sets of measurements are from the Locus MicroBenchmarks (TNC Performance Test Suite). The first line shows the time for a process to fork, the child to exit, and the parent to wait on the child. There is some slowdown in Solaris MC due to the additional overhead of going through the globalization layer and creating the virtual process object. The second line shows the time taken for a process that does multiple execs, either on a single node or between nodes. The Solaris MC overhead from local execs is minimal, since the virtual process object remains unchanged. There is additional overhead, however for a remote exec, due to

2. The programs used for these tests are available from <http://www.sunlabs.com/research/solaris-mc/process.html>.

| Operation                     | Solaris | Solaris MC | Solaris MC, remote |
|-------------------------------|---------|------------|--------------------|
| fork/exec (TNC #1)            | 9 ms    | 14 ms      | n/a                |
| (r)exec (TNC #5,#6)           | 32 ms   | 33 ms      | 58 ms              |
| fork/(r)exec/exec (TNC #7,#8) | 35 ms   | 42 ms      | 67 ms              |
| PIOCSTATUS ioctl on /proc     | 0.05ms  | 0.08 ms    | 3 ms               |
| Signals                       | 0.14 ms | 0.27 ms    | 2 ms               |

Table 4: Performance of various operations performed on standard Solaris, on Solaris MC locally, and on Solaris MC across nodes.

the process state (open file descriptors, etc.) that must be transmitted across the network, and the virtual process object that must be created on the remote node. The third line shows performance for a process that performs a cycle of fork, exec (local or remote), and exit. Again, Solaris MC has some overhead even for the local case due to the fork, and much more overhead for the remote exec. These measurements show that Solaris MC could use tuning of the virtual process object to improve fork performance, and remote exec performance is limited by the inter-node communication cost, but the globalization layer adds negligible cost to a local exec.

The next measurements show performance of the PIOCSTATUS ioctl on the /proc file system, which returns a 508 byte structure containing process status information. This measurement illustrates the performance of the globalized /proc file system and of the cross-node ioctl data copying. In the local case, there is a slowdown of about 30μs due to the overhead of going through the /proc globalization code and the PXFS ioctl layer. The remote case is considerably slower due to the network traffic. As discussed in Section 4.6, the copyout of ioctl data to user level is done through modified low-level copy functions. Thus, two network round-trips are required: one to invoke the remote ioctl, and a second to send the data back. Originally, an address-space object was created for every ioctl to handle data copying, but the overhead of object creation made even local ioctls take about 1 ms, an unacceptable overhead. However, by using a per-node object rather than a per-operation object, and by using the standard copy functions for local ioctls rather than the modified ones, the overhead was substantially reduced.

The final measurements in Table 4 shows performance for signalling between processes. In this test, two processes are started and send signals back and forth: the parent process sends a USR1 signal to the child process, the child process catches the signal and sends a USR1 signal to the parent, the parent catches the signal, and the cycle repeats. The time given is the average one-way time (i.e. half the cycle time). Again there is some

| Operating system, file system, process location        | Sequential      | Parallel        |
|--|-----------------|-----------------|
| Solaris, local UFS file system, local processes        | 26 sec (33 sec) | 18 sec (22 sec) |
| Solaris, remote NFS file system, local processes       | 36 sec (38 sec) | 23 sec (27 sec) |
| Solaris MC, local UFS file system, local processes     | 26 sec (32 sec) | 20 sec          |
| Solaris MC, local PXFS, local processes                | 27 sec (31 sec) | 19 sec          |
| Solaris MC, remote PXFS, local processes               | 29 sec (35 sec) | 21 sec          |
| Solaris MC, PXFS, processes running throughout cluster | 36 sec          | 15 sec          |

Table 5: Times for the compilation phase of the modified Andrew benchmark. Numbers in parentheses were measured on a 40 MHz node. The parallel measurements in the first five rows run multiple processes on the two processors on a node; the final parallel measurement runs processes on all nodes in the cluster.

slowdown for the local case in Solaris MC and a considerable overhead for between-node communication.

## 5.2 Parallel make performance

One interesting measure is how well system performance scales for parallel tasks on multiple nodes. Table 5 gives performance measurements for the system on the compilation phase of the modified Andrew benchmark [17], compiling a sequence of files sequentially and in parallel across the cluster under various conditions. The first two lines show compilation time for a single node running Solaris, accessing the files from a local disk or NFS, and compiling with sequential make or parallel make (which takes advantage of the dual processors on a node). Unfortunately, one of the nodes had 40 MHz processors, while the others had 50 MHz processors, which makes cluster measurements slightly harder to interpret; numbers in parentheses show results on the slower node. The next three lines show performance of a single node running Solaris MC, using a local disk running the UNIX file system, a local disk running the Solaris MC proxy file system (PXFS), or a remote node running the Solaris MC file system. Finally, the last line shows performance of Solaris MC when processes are executed around the cluster, either sequentially or in parallel.

Table 5 shows a speedup for parallel compilation on the Solaris MC cluster compared to a single Solaris node (15 sec vs. 18 sec), but this speedup isn't as dramatic as one would expect when going from 2 processors to 8 processors. Several factors account for this. First, the benchmark compile ends with a link phase that must be done sequentially and takes about 5 seconds. Thus, even with perfect scaling, the entire compile would take about 8 seconds. Second, there is a performance penalty in accessing PXFS files from a remote node (although comparing the PXFS measurements with the Solaris measurements shows that locally, PXFS is comparable to the native file system, and remotely it is faster than NFS). Thus, the compile is slowed down due to cross-

node file accesses. Third, the node with 40 MHz processors slows down the processes that are migrated to that node. Fourth, the parallel make uses a simple round-robin allocation policy, which yields non-optimum load balancing, especially when there is one slow node. Finally, comparing the last two lines of Table 5 shows that there is a significant performance penalty when processes are executed on remote nodes (increasing the time from 29 to 36 seconds for sequential execution). The effects of these factors can be reduced, however. Most significantly, with a faster interconnect and tuning of the object transport layer, the overheads due to communication between nodes will be reduced, improving file system and remote execution performance. With a faster interconnect and a balanced system, the performance improvement from parallel execution would be considerably better.

## 5.3 Performance discussion

Note that the Solaris operating system has been extensively tuned for performance, while Solaris MC is an almost entirely untuned research system. We expect these numbers to improve significantly with tuning, but there will still be some performance penalty due to the longer code path through the globalization layer.

Remote process operations are considerably slower than local operations because of the additional network transport time. The performance of distributed process management depends strongly on the performance of the underlying object system and the cluster network. We are in the process of optimizing the Solaris MC object framework to provide faster inter-node communication.

## 6 Related work

Several research and commercial operating systems provide distributed process management, such as Unisys Opus (Chorus) [4], Intel XP/S MP Paragon, OSF/1 AD TNC (Mach)[25], DCE TCF (Locus)[20], Sprite [7],

GLUnix [22], and MOSIX [3]. Solaris MC uses many of the concepts from these systems.

Process management in the Solaris MC operating system differs from previous systems in several ways. First, many of the previous systems build distributed process management from scratch; the Solaris MC OS demonstrates how distributed process management can be added to an existing commercial kernel while minimizing kernel changes. On the other hand, the Solaris MC OS provides a stronger single-system image than systems such as GLUnix, which build a globalization layer at user level on top of an existing kernel.

The Solaris MC OS also differs from previous systems in that it is built on an object-oriented communication framework, rather than a RPC-based framework. One key difference is that the object-oriented framework transparently routes invocations to the local or remote node as necessary, compared to RPC-based systems which require explicit marshalling of arguments and calling to a particular node. In addition, the object-oriented framework provides a built-in object reference counting mechanism; this avoids *ad hoc* mechanisms that are typically required in an RPC-based system to clean up state after node failures. The Solaris MC OS also illustrates how object-oriented programming can be added to an existing monolithic kernel.

The Solaris MC OS also presents new object-oriented interfaces to the process subsystem. These interfaces may be useful to applications that require more control over the process subsystem.

Finally, the Solaris MC OS shows how the `/proc` file system can be extended to a cluster to provide file access to process state throughout a cluster. Unlike the VPROCS [24] implementations of `/proc`, Solaris MC uses object inheritance to provide a distributed `/proc` through subclassing of the PXFS file system implementation.

## 7 Conclusions

Process management in the Solaris MC research operating system provides a distributed view of processes with a single `pid` space across a cluster of machines, while preserving POSIX semantics. It supports the standard UNIX process operations and the Solaris `/proc` file system as well as providing remote execution.

Process management is implemented in an object framework, with objects corresponding to each process, process group, and system node. This object framework simplified implementation of process management by providing transparent communication between nodes, failure notification, and reference counting.

Unlike the file system with the `vnode` interface, process management in UNIX generally lacks an interface for extending the system. While the `/proc` interface provides some control over processes, it is limited to status and process control operations. Process management in the Solaris MC operating system extends the access to the operating system's process internals by providing an object-oriented interface. In the future, the `/proc` interface and the object-oriented interface could be merged to provide a single extensible interface to the operating system's process management.

Process management was designed to allow most local operations to take place without network communication; there is no central server that must be contacted for process operations. There is some performance penalty due to the overhead of the globalization layer and due to creation of the virtual process object. With tuning, however, we expect this overhead to be reduced. Remote operations suffer a performance penalty due to the interconnect bandwidth and the overhead of the Solaris MC transport layer.

The main components in Solaris MC process management remaining to be implemented are process migration and load balancing, although we have remote process execution. Support for failure recovery and for full process group semantics also need to be implemented.

In conclusion, process management in the Solaris MC operating system illustrates how an existing monolithic operating system can be extended, with relatively few kernel changes, to provide transparent clusterwide process management. It also provides a set of object-oriented interfaces that can be used to provide access to the process internals.

## Acknowledgments

This work would not have been possible without the work of the entire Solaris MC team in building Solaris MC. The author is grateful to Scott Wilson, Yousef Khalidi, the anonymous referees and especially the paper "shepherd" Clem Cole for their helpful comments on the paper.

## References

- [1] AT&T, *Research Unix Version 8*, Murray Hill, NJ.
- [2] M. J. Bach, "Distributed file systems", *The Design of the UNIX Operating System*, Prentice-Hall, Englewood Cliffs, NJ, 1986.
- [3] A. Barak, S. Guday, and R. G. Wheeler, "The MOSIX Distributed Operating Systems," *Lecture Notes*

- in *Computer Science* 672, Springer-Verlag, Berlin, 1993.
- [4] N. Batlivala, *et al.*, "Experience with SVR4 Over CHORUS," *Proceedings of USENIX Workshop on Microkernels & Other Kernel Architectures*, April 1992.
- [5] J. Bernabeu, V. Matena, and Y. Khalidi, "Extending a Traditional OS Using Object-Oriented Techniques", 2nd Conference on Object-Oriented Technologies and Systems (COOTS), June 1996.
- [6] C. T. Cole, P. B. Flinn, and A. Atlas, "An Implementation of an Extended File System for UNIX," *Proceedings of Summer USENIX 1985*, pp. 131-144.
- [7] F. Douglass and J. Ousterhout, "Transparent Process Migration: Design Alternatives and the Sprite Implementation," *Software—Practice & Experience*, vol. 21(8), August 1991.
- [8] Fred Douglass, John K. Ousterhout, M. Frans Kaashoek, and Andrew S. Tanenbaum, "A Comparison of Two Distributed Systems: Amoeba and Sprite," *Computing Systems*, 4(4):353-384, Fall 1991.
- [9] R. Faulkner and R. Gomes, "The Process File System and Process Model in UNIX System V," *Proceedings of Winter Usenix 1991*.
- [10] G. Hamilton, M. L. Powell, and J. G. Mitchell, "Subcontract: A Flexible Base for Distributed Programming," *Symposium on Operating System Principles*, 1993, pp. 69-79.
- [11] M. Harchol-Balter and A. B. Downey, "Exploiting Process Lifetime Distributions for Dynamic Load Balancing," *Proceedings of ACM Sigmetrics '96 Conference on Measurement and Modeling of Computer Systems*, May 23-26 1996, pp 13-24.
- [12] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West, "Scale and Performance in a Distributed File System", *ACM Transactions on Computer Systems*, 6(1), February 1988, pp. 51-81.
- [13] IEEE, *IEEE Standard for Information Technology Portable Operating System Interface*, IEEE Std 1003.1b-1993, April 1994.
- [14] Y. Khalidi, J. Bernabeu, V. Matena, K. Shirriff, M. Thadani, "Solaris MC: A Multicomputer Operating System", *Proceedings of Usenix 1996*, January 1996, pp. 191-203.
- [15] Steven R. Kleiman, "Vnodes: An Architecture for Multiple File System Types in Sun UNIX", *Proceedings of Summer USENIX Conference 1986*, pp. 238-247.
- [16] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, Revision 1.2, December 1993.
- [17] J. Ousterhout, "Why Aren't Operating Systems Getting Faster As Fast As Hardware?" *Proceedings of Summer USENIX 1990*, pp. 247-256
- [18] J. Ousterhout, A. Cherenon, F. Douglass, M. Nelson, and B. Welch, "The Sprite Network Operating System," *IEEE Computer*, February 1988.
- [19] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey., and P. Winterbottom, "Plan 9 From Bell Labs," *Computing Systems*, Vol. 8, No. 3, Summer 1995, pp. 221- 254.
- [20] G. Popek and B. Walker, *The LOCUS Distributed System Architecture*, MIT Press, 1985.
- [21] Sun Microsystems, Inc. *IDL Programmer's Guide*, 1992.
- [22] Amin M. Vahdat, Douglas P. Ghormley, and Thomas E. Anderson, *Efficient, Portable, and Robust Extension of Operating System Functionality*, UC Berkeley Technical Report CS-94-842, December, 1994.
- [23] B. Walker, J. Lilienkamp, J. Hopfield, R. Zajcew, G. Thiel, R. Mathews, J. Mott, and F. Lawlor, "Extending DCE to Transparent Processing Clusters," *UniForum 1992 Conference Proceedings*, pp 189-199.
- [24] B. Walker, R. Zajcew, G. Thiel, *VPROCS: A Virtual Process Interface for POSIX systems*, Technical Report LA-0920, Locus Computing Corporation, May, 1992.
- [25] Roman Zajcew, *et al.*, "An OSF/1 UNIX for Massively Parallel Multicomputers," *Proceedings of Winter USENIX Conference 1993*.
- [26] R. Sandberg, D. Goldberg, D. Walsh, and B. Lyon, "The Design of the Sun Network File System," *Proceedings of Summer USENIX 1985*, pp 119-131.



# Adaptive and Reliable Parallel Computing on Networks of Workstations

Robert D. Blumofe  
Department of Computer Sciences  
The University of Texas at Austin  
Austin, Texas 78712  
rdb@cs.utexas.edu

Philip A. Lisiecki  
MIT Laboratory for Computer Science  
545 Technology Square  
Cambridge, Massachusetts 02139  
lisiecki@mit.edu

October 21, 1996

## Abstract

In this paper, we present the design of *Cilk-NOW*, a runtime system that adaptively and reliably executes functional *Cilk* programs in parallel on a network of UNIX workstations. *Cilk* (pronounced “silk”) is a parallel multithreaded extension of the C language, and all *Cilk* runtime systems employ a provably efficient thread-scheduling algorithm. *Cilk-NOW* is such a runtime system, and in addition, *Cilk-NOW* automatically delivers adaptive and reliable execution for a functional subset of *Cilk* programs. By adaptive execution, we mean that each *Cilk* program dynamically utilizes a changing set of otherwise-idle workstations. By reliable execution, we mean that the *Cilk-NOW* system as a whole and each executing *Cilk* program are able to tolerate machine and network faults. *Cilk-NOW* provides these features while programs remain *fault oblivious*, meaning that *Cilk* programmers need not code for fault tolerance. Throughout this paper, we focus on end-to-end design decisions, and we show how these decisions allow the design to exploit high-level algorithmic properties of the *Cilk* programming model in order to simplify and streamline the implementation.

## 1 Introduction

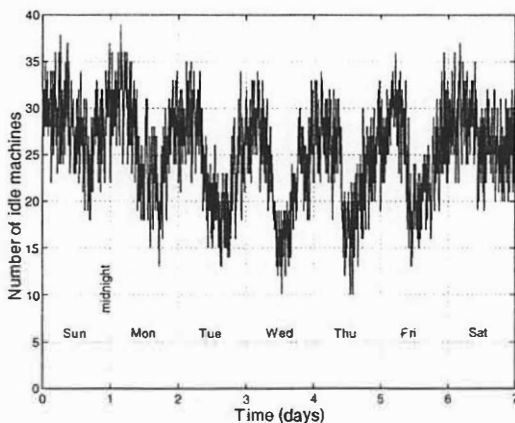
A strong case argues for the use of networks of workstations (NOWs) as parallel-computation platforms [3], and *Cilk-NOW* [6] is a software system that has been designed and implemented to run parallel programs easily and efficiently on networks of UNIX workstations. Implemented entirely in user-level software on top of UNIX, *Cilk-NOW* is a runtime system for a functional subset of the parallel *Cilk* language [6, 8, 26], a multithreaded extension of C. Applications written in *Cilk*

include graphics rendering, backtrack search, protein folding [37], and the *\*Socrates* chess program [25] which won second prize at the 1995 ICCA World Computer Chess Championship running on the 1824-node Intel Paragon at Sandia National Labs. Like all runtime systems for *Cilk*, *Cilk-NOW* schedules threads using a provably efficient algorithm based on the technique of random “work stealing” [6, 9] in which processors with no threads steal threads from victims chosen at random. With this algorithm, *Cilk* delivers performance that is guaranteed to be both efficient and predictable [6, 8]. In addition to thread scheduling, *Cilk-NOW* also performs *macroscheduling* [30]. That is, *Cilk-NOW* automatically identifies idle workstations and assigns those idle workstations to help out with running *Cilk* programs.

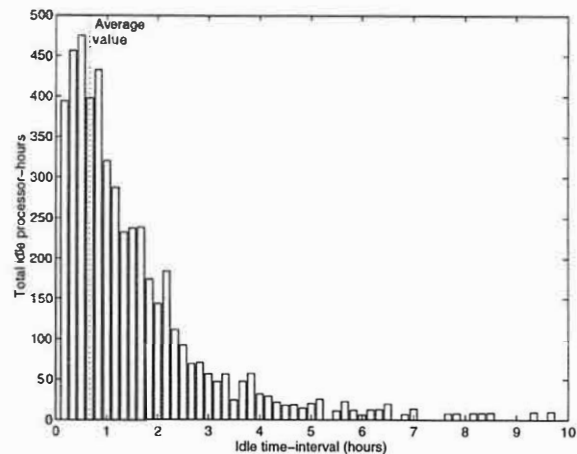
The *Cilk-NOW* runtime system is designed to execute *Cilk* programs efficiently in the highly dynamic environment of a NOW. Figure 1(a) plots the number of machines that were idle<sup>1</sup> at each point in time over the course of a typical week for a network of 50 SPARCstations at the MIT Laboratory for Computer Science. As can be seen from this plot, though more machines are idle at night, a significant number of machines are idle at various times throughout the day. Therefore, by adaptively using idle machines both day and night, we can take advantage of significantly more machine resources than if we run our parallel jobs as batch jobs during the night. Figure 1(b) is a histogram giving the total idle processor-hours broken down by idle time-interval, from this experiment. This histogram shows that a significant percentage of idle time (1104 processors-hours, or 19.1% of the total 5776 processor-hours) comes from machines that are idle for less than 30 minutes at a time. Thus, the efficient exploitation of idle machines requires that ma-

This research was supported in part by the Advanced Research Projects Agency (ARPA) under Grants N00014-94-1-0985 and N00014-92-J-1310. Robert Blumofe was supported in part by an ARPA High-Performance Computing Graduate Fellowship.

<sup>1</sup>For this experiment, a machine is idle if the keyboard and mouse have not been touched for 15 minutes and the 1, 5, and 15 minute processor load averages are below 0.35, 0.30, and 0.25 respectively. These load-average thresholds are reasonable but also somewhat arbitrary.



(a) Idle machines



(b) Idle time

**Figure 1:** (a) This plot shows the number of machines, out of the 50 machines in our network, that were idle at each point in time over the course of one typical week in March, 1995. (b) This histogram shows the number of idle processor-hours broken down by idle time-interval. When a machine remains idle for a period of  $t$  hours, it contributes  $t$  hours to the height of the bar plotted at position  $t$  rounded up to the nearest 10 minutes.

chines are able to join and leave a computation quickly and without human intervention. These observations are consistent with those of others [5, 20, 27, 28, 31].

Cilk-NOW provides the following features for running Cilk programs on a network of workstations.

**Ease of use.** A user can run a Cilk program in parallel on a NOW as if the program were only being run on the local workstation. The user simply types the program's command line, and then the Cilk-NOW runtime system automatically schedules the execution of the program in parallel across the network.

**Adaptive parallelism.** The Cilk-NOW system adaptively executes Cilk programs on a dynamically changing set of otherwise-idle workstations [6, 10]. When a given workstation is not being used by its owner, the workstation automatically joins in and helps out with the execution of a Cilk program. When the owner returns to work, the machine automatically retreats from the Cilk program.

**Fault tolerance.** The Cilk-NOW runtime system automatically performs checkpointing, detects failures, and performs recovery [6] while Cilk programs themselves remain *fault oblivious*. That is, Cilk-NOW provides fault tolerance without requiring that programmers code for fault tolerance.

**Flexibility.** The Cilk-NOW system allows the conditions that are used to determine the idleness of workstations to be set dynamically, in accordance with the tastes of the users and the owners of the machines whose cycles are being stolen. This flexibility preserves the sovereignty of each workstation's owner which is essential to ensure that owners are

willing to contribute their workstations for use by others.

**Security.** The Cilk-NOW system uses secure protocols that do not open a workstation to unauthorized users running foreign code on a machine. The desired degree of security is that which a given system uses to authenticate its remote execution protocol.

**Guaranteed performance.** The Cilk-NOW system executes Cilk programs using a work-stealing scheduler. This scheduler delivers performance that can be predicted accurately with a simple abstract model [6, 8]. Moreover this simple model can be adapted to the case of heterogeneous processors and networks [32].

Recently, we ran a Cilk protein-folding application `pfold` [37] using Cilk-NOW on a network of about 50 Sun SPARCstations connected by shared 10-Mb/s Ethernet to solve a large-scale protein-folding problem. The program ran for 9 days, surviving several machine crashes and reboots, utilizing 6566 processor-hours of otherwise-idle cycles, with no administrative effort on our part (besides typing `pfold` at the command-line to begin execution), while other users of the network went about their business unaware of the program's presence.

It is important to note that Cilk-NOW provides these features only for Cilk-2 programs which are essentially functional. Cilk-NOW does not support more recent versions of Cilk (Cilk-3 and Cilk-4) that incorporate virtual shared memory, and in particular, Cilk-NOW does not provide any kind of distributed shared memory. In addition, Cilk-NOW does not provide fault tolerance for its I/O facility.

In this paper, we present the design of Cilk-NOW, focusing on those features of Cilk-NOW that are particular to the NOW environment. The Cilk-2 language, work-stealing scheduler, MPP implementation, and guaranteed performance model have been covered at length in other papers [6, 8, 9, 26]. In this paper, we shall focus on adaptive parallelism and fault tolerance. Specifically, we will show how Cilk-NOW's end-to-end design [38] leverages algorithmic properties of the Cilk programming model and work-stealing scheduler in order to amortize all the overhead of adaptive parallelism and fault tolerance against the analytically and empirically bounded overhead of Cilk's work-stealing scheduler.

The remainder of this paper is organized as follows. In Section 2 we review the Cilk-2 language and work-stealing scheduler as first introduced in [8]. In Section 3 we describe the architecture of a Cilk job executing under the Cilk-NOW runtime system. Then, in Section 4 we explain how Cilk-NOW implements adaptive parallelism, and in Section 5 we explain how Cilk-NOW performs checkpointing, fault detection, and fault recovery. In Section 6 we describe the Cilk-NOW macroscheduling system architecture. In Section 7 we compare the Cilk-NOW system to related work. Finally, in Section 8 we outline plans for future work, and we conclude.

## 2 The Cilk language and work-stealing scheduler

In this section we overview the Cilk parallel multithreaded language and its runtime system's work-stealing scheduler [6, 8, 26]. For brevity, we shall not present the entire Cilk language, and we shall omit some details of the work-stealing algorithm. Since Cilk-2 forms the basis for the Cilk-NOW system, we shall focus on the Cilk-2 language and on the Cilk-2 runtime system as implemented without adaptive parallelism or fault tolerance.

A Cilk program contains one or more *Cilk procedures*, and each Cilk procedure contains one or more *Cilk threads*. A Cilk procedure is the parallel equivalent of a C function, and a Cilk thread is a nonsuspending piece of a procedure. The Cilk runtime system manipulates and schedules the threads. The runtime system is not aware of the grouping of threads into procedures. Cilk procedures are purely an abstraction supported by the `cilk2c` type-checking preprocessor [33].

Consider a program that uses double recursion to compute the Fibonacci function. The Fibonacci function  $\text{fib}(n)$  for  $n \geq 0$  is defined as

$$\text{fib}(n) = \begin{cases} n & \text{if } n < 2; \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{otherwise.} \end{cases}$$

```
thread Fib (cont int k, int n)
{ if (n<2)
  send.argument (k, n);
  else
  { cont int x, y;
    spawn.next Sum (k, ?x, ?y);
    spawn Fib (x, n-1);
    spawn Fib (y, n-2);
  }
}

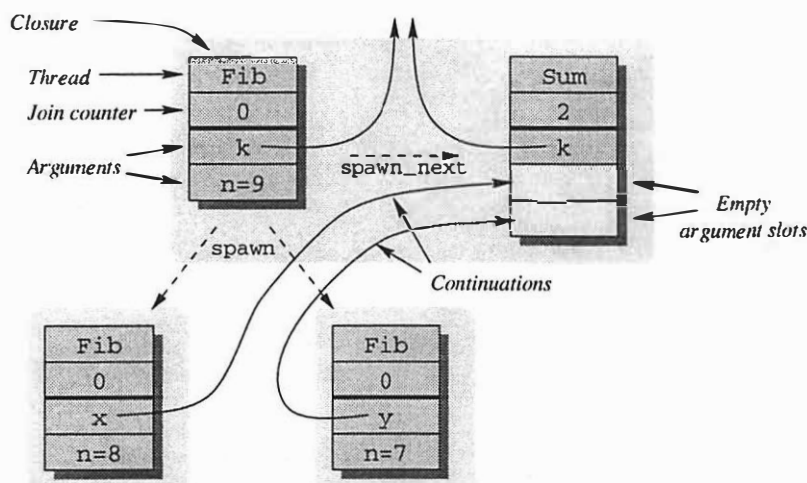
thread Sum (cont int k, int x, int y)
{ send.argument (k, x+y);
}
```

**Figure 2:** A Cilk procedure to compute the  $n$ th Fibonacci number. This procedure contains two threads, *Fib* and *Sum*.

Figure 2 shows how this function is written as a Cilk procedure consisting of two Cilk threads: *Fib* and *Sum*. While double recursion is a terrible way to compute Fibonacci numbers, this toy example does illustrate a common pattern occurring in divide-and-conquer applications: recursive calls solve smaller subcases and then the partial results are merged to produce the final result.

A Cilk thread generates parallelism at runtime by *spawning* a child thread that is the *initial thread* of a child procedure. A spawn is the parallel equivalent of a function call. A spawn differs from a call in that when a thread spawns a child, the parent and child may execute concurrently. After spawning one or more children, the parent thread cannot then wait for its children to return—in Cilk, threads never suspend. Rather, the parent thread must additionally spawn a *successor thread* to wait for the values “returned” from the children. The spawned successor is part of the same procedure as its predecessor. The child procedures return values to the parent procedure by sending those values to the parent's waiting successor. Thus, a thread may wait to begin executing, but once it begins executing, it cannot suspend. This style of interaction among threads is called *continuation-passing style* [4]. Spawning successor and child threads is done with the `spawn.next` and `spawn` keywords respectively. Sending a value to a waiting thread is done with the `send.argument` statement. The Cilk runtime system implements these primitives using two basic data structures: closures and continuations.

*Closures* are data structures employed by the runtime system to keep track of and schedule the execution of spawned threads. Whenever a thread is spawned, the runtime system allocates a closure for it from a simple heap. A closure consists of a pointer to the code for that thread, a slot for each of the thread's specified arguments, and a *join counter* indicating the number of missing arguments that need to be supplied before the thread is ready to run. The closure, or equivalently the spawned thread, is *ready*



**Figure 3:** The `Fib` thread spawns a successor and two children. For the successor, it creates a closure with 2 empty argument slots, and for each child, it creates a closure with a continuation referring to one of these empty slots. The background shading denotes Cilk procedures.

if it has obtained all of its arguments, and it is *waiting* if some arguments are missing. To run a ready closure, the Cilk scheduler invokes the thread using the values in the closure as arguments. When the thread dies, the closure is freed.

A *continuation* is a global reference to an empty argument slot of a closure, implemented as a compound data structure containing a pointer to a closure and an offset that designates one of the closure's argument slots. Continuations are typed with the C data type of the slot in the closure. In the Cilk language, continuations are declared by the type modifier keyword `cont`. For example, the `Fib` thread declares two integer continuations, `x` and `y`.

Using the `spawn_next` primitive, a thread spawns a successor thread by creating a closure for the successor. The successor thread is part of the same procedure as its predecessor. For example, in the `Fib` thread, the statement `spawn_next Sum (k, ?x, ?y)` allocates a closure with `Sum` as the thread and three argument slots, as illustrated in Figure 3. The first slot is initialized with the continuation `k` and the last two slots are empty. The continuation variables `x` and `y` are initialized to refer to these two empty slots, and the join counter is set to 2. This closure is waiting.

Similarly, using the `spawn` primitive, a thread spawns a child thread by creating a closure for the child. The child thread is the initial thread of a newly spawned child procedure. The `spawn` statement is semantically identical to `spawn_next`. For example, the `Fib` thread spawns two children as shown in Figure 3. The statement `spawn Fib (x, n-1)` allocates a closure with `Fib` as the thread and two argument slots. The first slot is initialized with the continuation `x` which, as a consequence of the previous statement, refers to a slot in its parent's successor closure. The second slot is initialized

with the value of `n-1`. The join counter is set to zero, so the thread is ready.

An executing thread sends a value to a waiting thread by placing the value into an argument slot of the waiting thread's closure. The `send_argument` statement sends a value to the empty argument slot of a waiting closure specified by its argument. The types of the continuation and the value must be compatible. The join counter of the waiting closure is decremented, and if it becomes zero, then the closure is ready. For example, the statement `send_argument (k, n)` in `Fib` writes the value of `n` into an empty argument slot in the parent procedure's waiting `Sum` closure and decrements its join counter. When the `Sum` closure's join counter reaches zero, it is ready. When the `Sum` thread gets executed, it adds its two arguments, `x` and `y`, and then uses `send_argument` to "return" this result up to its parent procedure's waiting `Sum` thread.

At runtime, each processor maintains a "ready" deque (double-ended queue) which contains all of the ready closures. Whenever a closure is created, if its join counter is 0, then it is placed on the head of the ready deque. Whenever a `send_argument` call is made, the join counter is decremented, and if the join counter is decremented to zero, then the closure is placed on the head of the ready deque. When a thread finishes, the next thread to execute is chosen from the head of the ready deque.

If no threads are available in the ready deque, a processor engages in *work stealing*. To steal work, a processor, called the *thief*, chooses another processor, called the *victim*, at random and requests a closure to be sent back. If that processor has any closures in its ready deque, one is removed from the tail of the victim's ready deque and sent across the network to the thief, who will add this

closure to its own ready deque. The thief may then begin work on the stolen closure. If the victim has no ready closures, it informs the thief who then tries to steal from another random processor until a ready closure is found or program execution completes.

This simple work-stealing scheduler has been shown, both analytically and empirically, to deliver efficient and predictable performance [6, 8, 9] for “well structured” computations. A *well structured* computation is one in which each procedure sends values (with `send.argument`) only to its parent and only as the last action performed by its last thread. For well structured computations executing on any number  $P$  of processors, the execution time can be modeled accurately as  $T_1/P + T_\infty$  where  $T_1$  denotes the *work* of the computation—that is, the execution time with 1 processor—and  $T_\infty$  denotes the *critical-path length*—that is, the theoretical execution time on an ideal machine with infinitely many processors. Such performance is within a factor of 2 of optimal, and additionally when the critical path is short compared to the amount of work per processor, such performance displays *linear speedup*.

The key element in proving this  $T_1/P + T_\infty$  performance bound is the fact that closures are always stolen from the tail of the ready deque. For well structured computations, a closure that is on the critical path must be at the tail of some processor’s ready deque. Thus, when processors are not executing closures, they are stealing work and, therefore, are likely to be making progress on the critical path. As a corollary to this result, the number of work-steal attempts per processor is proportional to the critical-path length and does not grow with the work. Thus, a computation with a sufficiently short critical path compared to the work per processor can continue to display linear speedup even when communication is very expensive. This idea of amortizing overhead against the critical path plays an important role in our later discussion of adaptive parallelism and fault tolerance.

### 3 Cilk-NOW job architecture

The Cilk-NOW runtime system consists of several component programs that (in addition to macroscheduling duties discussed later) manage the execution of each individual Cilk program. In this section, we shall cover the architecture of a Cilk program as it is executed by the Cilk-NOW runtime system, explaining the operation of each component and their interactions.

In Cilk-NOW terminology, we refer to an executing Cilk program as a *Cilk job*. Since Cilk programs are parallel programs, a Cilk job consists of several processes running on several machines. One process, called the *clearinghouse*, in each Cilk job runs a system-supplied program called `CilkChouse` that is responsible for

keeping track of all the other processes that comprise a given job. These other processes are called *workers*. A worker is a process running the actual executable of a Cilk program. Since Cilk jobs are adaptively parallel, the set of workers is dynamic. At any given time during the execution of a job, a new worker may join the job or an existing worker may leave. Thus, each Cilk job consists of one or more workers and a clearinghouse to keep track of them.

The Cilk-NOW runtime system contains additional components that perform macroscheduling as discussed in Section 6, but for the purpose of our present discussion, we need only introduce the “node managers.” A *node manager* is a process running a system-supplied program called `CilkNodeManager`. A node manager runs as a background daemon on every machine in the network. It continually monitors its machine to determine when the machine is idle.

To see how all of these components work together in managing the execution of a Cilk job, we shall run through an example. (In describing interactions with the macroscheduler, we shall refer to the macroscheduler as a single entity, though actually, as we shall see in Section 6, the macroscheduler is a distributed subsystem with several components.) Suppose that a user sits down at a machine called Penguin to run the `pfold` program. In our example, the user types

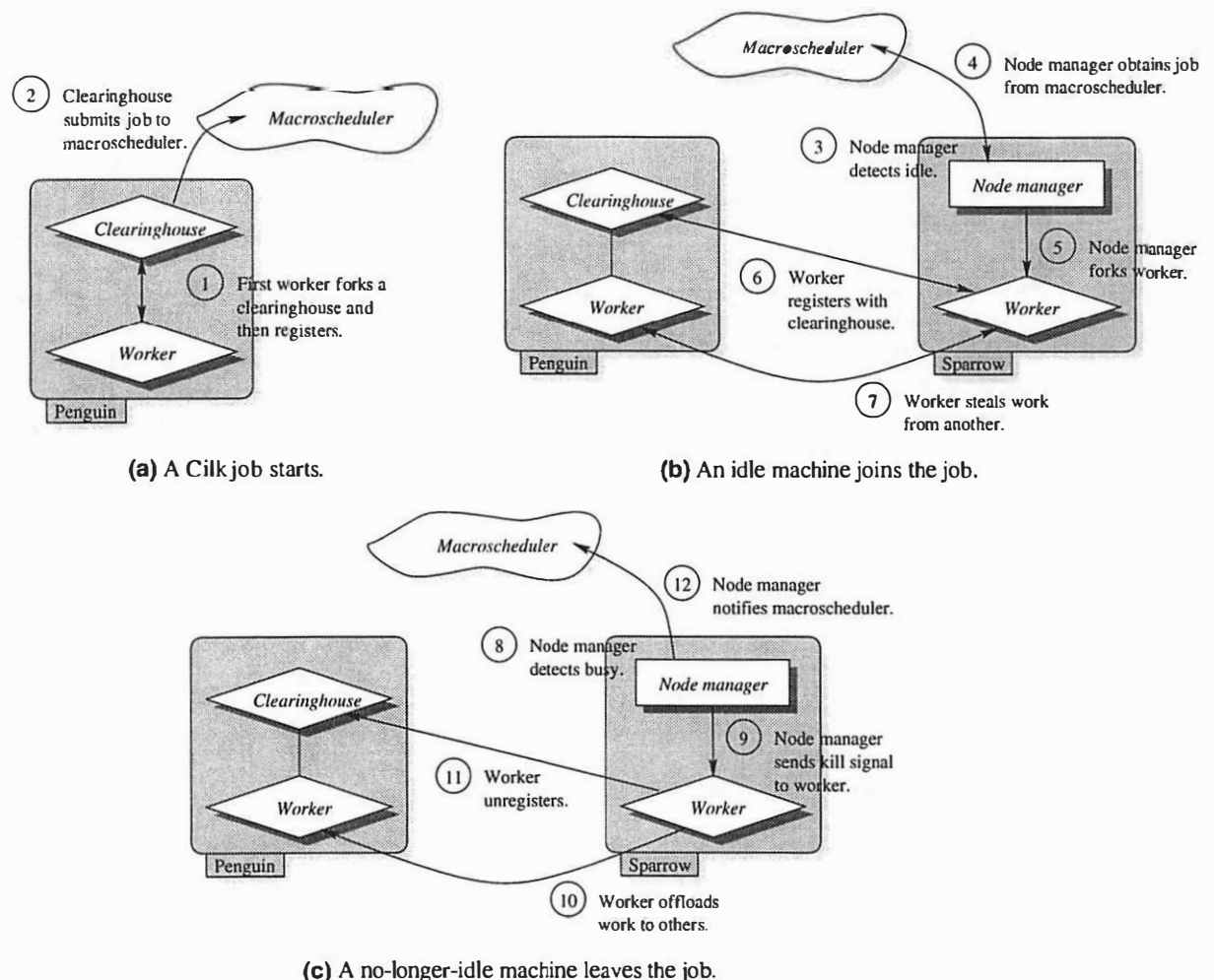
```
pfold 3 7
```

at the shell, thereby launching a Cilk job to enumerate all protein foldings using 3 initial folding sequences and starting with the 7th one.

The new Cilk job begins execution as illustrated in Figure 4(a). The new process running the `pfold` executable is the first worker and begins execution by forking a clearinghouse with the command line

```
CilkChouse -- pfold 3 7.
```

Thus, the clearinghouse knows that it is in charge of a job whose workers are running “`pfold 3 7`.” The clearinghouse begins execution by sending a *job description* to the macroscheduler. The job description is a record containing several fields. Among these fields is the name of the Cilk program executable—in this case `pfold`—and the clearinghouse’s network address. The clearinghouse then goes into a service loop waiting for messages from its workers. After forking the clearinghouse, the first worker *registers* with the clearinghouse by sending it a message containing its own network address. Now the clearinghouse knows about one worker, and it responds to that worker by assigning it a unique *name*. Workers are named with numbers, starting with number 0. Having registered, worker 0 begins executing the Cilk program as described in Section 2. We now have a running Cilk job with one worker.



**Figure 4:** (a) The first worker forks a clearinghouse, and then the clearinghouse submits the job to the macroscheduler. (b) When the node manager detects that its machine is idle, it obtains a job from the macroscheduler and then forks a worker. The worker registers with the clearinghouse and then begins work stealing. (c) When the node manager detects that its machine is no-longer idle, it sends a kill signal to the worker. The worker catches this signal, offloads its work to other workers, unregisters with the clearinghouse, and then terminates.

A second worker joins the Cilk job when some other workstation in the network discovers that it is idle, as illustrated in Figure 4(b). Suppose the node manager on a machine named Sparrow detects that the machine is idle. The node manager sends a message to the macroscheduler, and the macroscheduler responds with the job description of a Cilk job for the machine to work on. In this case, the job description specifies our `pfold` job by giving the name of the executable—`pfold`—and the network address of the clearinghouse. The node manager then uses this information to fork a new worker as a child with the command line

```
pfold -NoChouse
-Address=clearinghouse-address
--
```

The `-NoChouse` flag on the command line tells the

worker that it is to be an additional worker in an already existing Cilk job. (Without this flag, the worker would fork a new clearinghouse and start a new Cilk job.) The `-Address` field on the command line tells the worker where in the network to find the clearinghouse. The worker uses this address to send a registration message, containing its own network address, to the clearinghouse. The clearinghouse responds with the worker's assigned name—in this case, number 1—and the job's command-line arguments—in this case, "`pfold 3 7`." Additionally, the clearinghouse responds with a list of the network addresses of all other registered workers. Now the new worker knows the addresses of the other workers, so it can commence execution of the Cilk program and steal work as described in Section 2. We now have a running Cilk job with two workers.

Now, suppose that someone touches the keyboard on Sparrow. In this case, the node manager detects that the machine is busy, and the machine leaves the Cilk job as illustrated in Figure 4(c). After detecting that the machine is busy, the node manager sends a kill signal to its child worker. The worker catches this signal and prepares to leave the job. First, the worker offloads all of its closures to other workers as explained in more detail in Section 4. Next, the worker sends a message to the clearinghouse to *unregister*. Finally, the worker terminates.

When a Cilk job is running, each worker periodically checks in with the clearinghouse. Specifically, each worker periodically (every 2 seconds) sends a message to the clearinghouse, and the clearinghouse responds with an *update* message informing the worker of any other workers that have left the job and any new workers that have joined the job. For each new worker that has joined, the clearinghouse also provides the network address. If the clearinghouse does not receive any messages from a given worker for an extended period of time (30 seconds), then the clearinghouse determines that the worker has crashed. In later update messages, the clearinghouse informs the other workers of the crash, and the other workers take appropriate remedial action as described in Section 5.

All communication between workers, and between workers and the clearinghouse, is implemented with UDP/IP [13, 40]. Knowing that UDP datagrams are unreliable, the Cilk-NOW protocols incorporate appropriate mechanisms, such as acknowledgments, retries, and timeouts, to ensure correct operation when messages get lost. We shall not discuss these mechanisms in any detail, and in order to simplify our exposition of Cilk-NOW, we shall often speak of messages being sent and received as if they are reliable. What we will say about these mechanisms is that they are built on top of UDP but without any effort to create a reliable message-passing layer. Rather these mechanisms are built directly into the runtime system's protocols, so in the common case when a message does get through, Cilk-NOW pays no overhead to make the message reliable.

We chose to build Cilk-NOW's communication protocols using an unreliable message-passing layer instead of a reliable one for two reasons, both based on end-to-end design arguments [38]. First, reliable layers such as TCP/IP [40] and PVM [41] perform implicit acknowledgments and retries to achieve reliability. Therefore, such layers either preclude the use of asynchronous communication or require extra buffering and copying. A layer such as UDP which provides minimal service guarantees can be implemented with considerably less software overhead than a layer with more service features. In the common case when the additional service is not needed, the minimal layer can easily outperform its fully-

featured counterpart. Second, in an environment where machines can crash and networks can break, the notion of a "reliable" message-passing layer is somewhat suspect. A runtime system operating in an inherently unreliable environment cannot expect the message-passing layer to make the environment reliable. Rather, the runtime system must incorporate appropriate mechanisms into its protocols to take action when a communication endpoint or link fails. For these reasons, we chose to build the Cilk-NOW runtime system on top of a minimal layer of message-passing service and incorporate mechanisms directly into the runtime system's protocols in order to handle issues of reliability. The downside to this approach is complexity. The protocols implemented in the Cilk-NOW runtime system are complex: the code for these protocols takes almost 20 percent of the total runtime-system code, and the programming effort was probably near half of the total. Nevertheless, this was a one-time effort that we expect will reap performance rewards for a long time to come.

## 4 Adaptive parallelism

Adaptive parallelism allows a Cilk job to take advantage of idle machines whether or not they are idle when the job starts and whether or not they will remain idle for the duration of the job. In order to efficiently utilize machines that may join and leave a running job, the overhead of supporting this feature must not excessively slow down the work of any worker at a time when it is not joining or leaving. As we saw in the previous section, a new worker joins a job easily enough by registering with the clearinghouse and then stealing a closure. A worker leaves a job by migrating all of its closures to other workers, and here the danger lies. When we migrate a waiting closure, other closures with continuations that refer to this closure must somehow update these continuations so they can find the waiting closure at its new location. (Without adaptive parallelism, waiting closures never move.) Naively, each migrated waiting closure would have to inform every other closure of its new location. In this section, we show how we can take advantage of Cilk's well structuring and the work-stealing scheduler to make this migration extremely simple and efficient. (Experimental results documenting the efficiency of Cilk-NOW's adaptive parallelism have been omitted for lack of space but can be found in [6].)

Our approach is to impose additional structure on the organization of closures and continuations, such that the structure is cheap to maintain while simplifying the migration of closures. Specifically, we maintain closures in "subcomputations" that migrate en masse, and every continuation in a closure refers to a closure in the same subcomputation. In order to send a value from a clo-

sure in one subcomputation to a closure in another, we forward the value through intermediate “result closures,” and give each result closure the ability to send the value to precisely one other closure in one other subcomputation. With this structure and these mechanisms, all of the overhead associated with adaptive parallelism (other than the actual migration of closures) occurs only when closures are stolen, and as we saw in Section 2, the number of steals grows at most linearly with the critical path of the computation and is not a function of the work. The bulk of this section’s exposition concerns the organization of closures in subcomputations and the implementation of continuations. After covering these topics, the mechanism by which closures are migrated to facilitate adaptive parallelism is quite straightforward.

In Cilk-NOW, every closure is maintained in one of three pools associated with a data structure called a *subcomputation*. A subcomputation is a record containing (among other things) three pools of closures. The *ready pool* is the list of ready closures described in Section 2. The *waiting pool* is a list of waiting closures. The *assigned pool* is a list of ready closures that have been stolen away. Program execution begins with one subcomputation—the *root* subcomputation—allocated by worker 0 and containing a single closure—the initial thread of `cilkmain`—in the ready pool. In general, a subcomputation with any closures in its ready pool is said to be *ready*, and ready subcomputations can be executed by the scheduler as described in Section 2 with the additional provision that each waiting closure is kept in the waiting pool and then moved to the ready pool when its join counter decrements to zero. The assigned pool is used in work stealing as we shall now see.

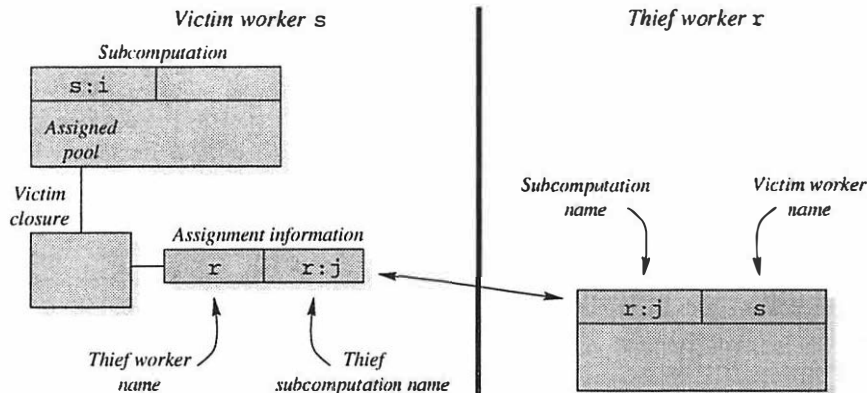
The act of work stealing creates a new subcomputation on the thief which is linked to a copy of the stolen closure kept in an assigned pool on the victim. If a worker needs to steal work, then before sending a steal request to a victim, it allocates a new subcomputation from a simple runtime heap and gives the subcomputation a unique *name*. The subcomputation’s name is formed by concatenating the thief worker’s name and a number unique to that worker. The first subcomputation allocated by a worker  $r$  is named  $r : 1$ , the second is named  $r : 2$ , and so on. The root subcomputation is named  $0 : 1$ . The steal request message contains the name of the thief’s newly allocated subcomputation. When the victim worker gets the request message, if it has any ready subcomputations, then it chooses a ready subcomputation in round-robin fashion, removes the closure at the tail of the subcomputation’s ready pool, and places this *victim closure* in the assigned pool. As illustrated in Figure 5, the victim worker then *assigns* the closure to the thief’s subcomputation by adding to the closure an *assignment information* record allocated from a simple runtime heap, and then

storing the name of the thief worker and the name of the thief’s subcomputation (as contained in the steal request message) in the assignment information. Finally, the victim worker sends a copy of the closure to the thief. When the thief receives the stolen closure, it records the name of the victim worker in its subcomputation, and it places the closure in the subcomputation’s ready pool. Now the thief’s subcomputation is ready, and the thief worker may commence executing it. Notice that the victim closure and thief subcomputation can refer to each other via the thief subcomputation’s name which is stored both in the victim closure’s assignment information and in the thief subcomputation, as illustrated in Figure 5.

When a worker finishes executing a subcomputation, the link between the subcomputation and its victim closure is destroyed. Specifically, when a subcomputation has no closures in any of its three pools, then the subcomputation is *finished*. A worker with a finished subcomputation sends a message containing the subcomputation’s name to the subcomputation’s victim worker. Using this name, the victim worker finds the victim closure. This closure is removed from its subcomputation’s assigned pool and then the closure and its assignment information are freed. The victim worker then acknowledges the message, and when the thief worker receives the acknowledgment, it frees its subcomputation. When the root subcomputation is finished, the entire Cilkjob is finished.

In addition to allocating a new subcomputation, whenever a worker steals a closure, it also allocates a new “result” closure, and it alters the continuation in the stolen closure so that it refers to the result closure. Consider a thief stealing a closure, and suppose the victim closure contains a continuation referring to a closure that we call the *target*. (The victim and target closures must be in the same subcomputation in the victim worker.) Continuations are implemented as the address of the target closure concatenated with the index of an argument slot in the target closure. Therefore, the continuation in the victim closure contains the address of the target closure, and this address is only meaningful to the victim worker. Thus, when the thief worker receives the stolen closure, it replaces the continuation with a new continuation referring to an empty slot in a newly allocated *result* closure. The stolen and result closures are part of the same subcomputation. The result closure’s thread is a special system thread whose operation we shall explain shortly. This thread takes one argument: a *result value*. The result value is initially missing, and the continuation in the stolen closure is set to refer to this argument slot. The result closure is waiting and its join counter is 1.

Using continuations to send values from one thread to another operates as described in Section 2, but when a value is sent to a result closure, communication between



**Figure 5:** A victim closure stolen from the subcomputation  $s:i$  of victim worker  $s$  is assigned to the thief subcomputation  $r:j$ . The victim closure is placed in the assigned pool and augmented with assignment information that records the name of the thief worker and the name of the thief subcomputation. The thief subcomputation records its own name and the name of the victim worker. Thus, the victim closure and thief subcomputation can refer to each other via the thief subcomputation's name.

different subcomputations occurs. When a result closure receives its result value, it becomes ready, and when its thread executes, it forwards the result value to another closure in another subcomputation as follows. When a worker executing a subcomputation executes a result closure's thread, it sends a message to the subcomputation's victim worker. This message contains the subcomputation's name as well as the result value that is the thread's argument. When the victim worker receives this message, it uses the subcomputation name to find the victim closure, and then it uses the continuation in the victim closure to send the result value to the target.

To summarize, each subcomputation contains a collection of closures and every continuation in a closure refers to another closure in the same subcomputation. To send a value from a closure in one subcomputation to a closure in another, the value must be forwarded through an intermediate result closure.

With this structure, migrating a subcomputation from one worker to another is fairly straightforward. At the source worker, the entire subcomputation is "pickled" by giving each of the subcomputation's closures a number and replacing each continuation's pointer with the corresponding number. Then, after sending the closures to the destination worker, the destination worker reconstructs the subcomputation by reversing the pickling operation. The subcomputation keeps its name, so after a migration, the first component of the subcomputation name will be different than the name of the worker. When the subcomputation and all of its closures have been migrated to their destination worker, this worker sends a message to the subcomputation's victim worker to inform the victim closure of its thief subcomputation's new thief worker. Additionally, for each of the subcomputation's assigned closures, it sends a message to the thief worker to inform the thief subcomputation of its victim closure's new vic-

tim worker. Thus, all of the links between victim closures and thief subcomputations are restored.

## 5 Fault tolerance

With transparent fault tolerance built into the Cilk-NOW runtime system, Cilk jobs may survive machine crashes or network outages despite the fact that Cilk programs are fault oblivious, having been coded with no special provision for handling machine or network failures. If a worker crashes, then other workers automatically redo any work that was lost in the crash. In the case of a more catastrophic failure, such as a power outage, a total network failure, or a crash of the file server, then all workers may crash. For this case, Cilk-NOW provides automatic checkpointing, so when service is restored, the Cilk job may be restarted with minimal lost work. Recall that Cilk-NOW does not provide fault tolerance for I/O.

In this section, we show how the structure used to support adaptive parallelism—which leverages Cilk's tree structure and the work-stealing scheduler—may be further leveraged to build these fault tolerant capabilities in Cilk-NOW. As with adaptive parallelism, all of the overhead associated with fault tolerance (other than the cost of periodic checkpoints) can be amortized against the number of steals which grows at most linearly with the critical path and is not a function of the work.

Given adaptive parallelism, fault tolerance is only a short step away. With adaptive parallelism, a worker may leave a Cilk job, but before doing so, it first migrates all of its subcomputations to other workers. In contrast, when a worker crashes, all of its subcomputations are lost. To support fault tolerance, we add a mechanism that allows surviving workers to redo any work that was done by the lost subcomputations. Such a mechanism must address two fundamental issues. First, not all work

is necessarily idempotent, so redoing work may present problems. We address this issue with a technique that we call a *return transaction*. Specifically, we ensure that the work done by any given subcomputation does not affect the state of any other subcomputations until the given subcomputation finishes. Thus, from the point-of-view of any other subcomputation, the work of a subcomputation appears as a transaction: either the subcomputation finishes and commits its work by making it visible to other subcomputations, or the subcomputation never happened. Second, the lost subcomputations may have done a large amount of work, and we would like to minimize the amount of work that needs to be redone. We address this issue by incorporating a transparent and fully distributed checkpointing facility. This checkpointing facility also allows a Cilk job to be restarted in the case of a total system failure in which every worker crashes.

To turn the work of a subcomputation into a return transaction, we modify the behavior of the subcomputation's result closure. In Cilk, returning a value is always the last operation performed by a Cilk procedure, so the result closure cannot be ready until the subcomputation is finished. In addition, recall that the execution of the result closure and the finishing of the subcomputation both warrant a message to the victim worker. Thus, we bundle these two messages into a single larger message sent to the victim worker. When the victim worker receives this message, it commits all of the thief subcomputation's work by sending the appropriate result value from the victim closure, freeing the victim closure (and its assignment information), and sending an acknowledgment back to the thief worker.

With subcomputations having this transactional nature, a Cilk job can tolerate individual worker crashes as follows. Suppose a worker crashes. Eventually, the clearinghouse will detect the crash, and the other living workers will learn of the crash at the next update from the clearinghouse. When a worker learns of a crash, it goes through all of its subcomputations, checking each assigned closure to see if it is assigned to the crashed worker. Each such closure is moved from the assigned pool back to the ready pool (and its assignment information is freed). Thus, all of the work done by the closure's thief subcomputation which has been lost in the crash will eventually be redone. Additionally, when a worker learns of a crash, it goes through all of its subcomputations to see if it has any that record the crashed worker as the subcomputation's victim. For each such subcomputation, the worker aborts it as follows. The worker goes through all of the subcomputation's assigned closures sending to each thief worker an *abort* message specifying the name of the thief subcomputation. Then the worker frees the subcomputation and all of its closures. When a worker receives an abort message, it finds the

thief subcomputation named in the message and recursively aborts it. All of the work done by these aborted subcomputations must eventually be redone. In order to avoid aborting all of these subcomputations (which may comprise the entire job in the case when the root subcomputation is lost) and redoing potentially vast amounts of work, and in order to allow restarting when the entire job is lost, we need checkpointing.

Cilk-NOW performs automatic checkpointing without any synchronization among different workers and without any notion of global state. Specifically, each subcomputation is periodically checkpointed to a file named with the subcomputation's name. For example, a subcomputation named `r:i` would be checkpointed to a file named `scomp_r.i`. We assume that all workers in the job have access to a common file system (through NFS or AFS, for example), and all checkpoint files are written to a common checkpoint directory.<sup>2</sup> To write a checkpoint file for a subcomputation `r:i`, the worker first opens a file named `scomp_r.i.temp`. Then, it writes the subcomputation record and all of the closures—including the assignment information for the assigned closures—into the file. Finally, it atomically renames the file `scomp_r.i.temp` to `scomp_r.i`, overwriting any previous checkpoint file. A checkpoint file can be read to recover the subcomputation. On writing a checkpoint file, the worker additionally prunes any no-longer-needed checkpoint files.

If workers crash, the lost subcomputations can be recovered from checkpoint files. In the case of a single worker crash, the lost subcomputations can be recovered automatically. When a surviving worker finds that it has a subcomputation with a closure assigned to the crashed worker, then it can recover the thief subcomputation by reading the checkpoint file. In the case of a large-scale failure in which every worker crashes, the Cilk job can be restarted from checkpoint files by setting the `-Recover` flag on the command line. Recovery begins with the root subcomputation whose checkpoint file is `scomp_0.1`. After recovering the root subcomputation, then every other subcomputation can be recovered by recursively recovering the thief subcomputation for each of the root subcomputation's assigned closures.

## 6 Cilk-NOW macroscheduling

The Cilk-NOW runtime system contains components that perform macroscheduling [30]. The macroscheduler identifies idle machines and determines which machines work on which jobs. In this section, we discuss each component of the macroscheduler, and we show how

<sup>2</sup>We have not yet implemented any sort of distributed file system. In the current implementation, workers implicitly synchronize when they write checkpoint files, since they all access a common file system.

they work together.

Like the workers and the clearinghouse, which together comprise a single parallel Cilk job, the components of the macroscheduler are distributed across the network. As already mentioned, each machine in the network runs a node manager, an instance of the `CilkNodeManager` program, that monitors the machine's idleness and the status of a worker if one is present. In addition to the clearinghouse, each Cilk job executes a single *job manager*, an instance of the `CilkJobManager` program, that services requests for the job description. Each of these components registers with a central *job broker*, an instance of the `CilkJobBroker` program. The job broker keeps track of the set of node managers and job managers running in the network. The Cilk-NOW runtime system can continue operation even if some of these components, including the job broker, fail.

Each machine in the network runs a node manager that is responsible for determining when the machine is idle. When the machine is being used, the node manager wakes up every 5 seconds to determine if the machine has gone idle. It looks at how much time has elapsed since the keyboard and mouse have been touched, the number of users logged in, and the processor load averages. The node manager then passes these values through a predicate to decide if the machine is idle. A typical predicate might require that the keyboard and mouse have not been touched for at least 5 minutes and the 1-minute processor load average is below 0.35. The predicate can be customized for each machine. We believe that maintaining the owner's sovereignty is essential if we want owners to allow their machines to be used for parallel computation. A user can change a predicate with a simple command-line utility called `CilkPred`. For example, issuing the command

```
CilkPred user=lisiecki global add  
idletime=900
```

causes any workstation on the network to require that the user "lisiecki" be idle for at least 900 seconds. Alternatively, a user might issue the command

```
CilkPred always node=vulture add  
load=.2,.2,.2
```

which applies only to the workstation `Vulture` and requires it to have a load average of 0.2 or less for all of the 1, 5, and 15 minute load averages.

When all applicable conditions of the predicate are satisfied, the machine is idle, and the node manager obtains a job description from a job manager (using an address given by the job broker or another node manager) and forks a worker. The node manager then monitors the worker and continues to monitor the machine's idleness. With a worker running, the node manager wakes

up once every second to determine if the machine is still idle (adding an estimate of the running job's processor usage to any processor load-average threshold). If the machine is no longer idle, then the node manager sends a kill signal to the worker as previously described. When the worker process dies for any reason, the node manager takes one of two possible actions. If the machine is still idle, then it obtains a new job description and forks a new worker. If the machine is no longer idle, then it returns to monitoring the machine once every 5 seconds.

When a Cilk job begins execution, a job manager is started automatically by the clearinghouse. The clearinghouse submits the job description to the job manager as alluded to in Section 3, and then the job manager registers itself with the job broker. The job manager then goes to sleep, and it periodically wakes up to reregister with the job broker in case the job broker has crashed and restarted. When the job terminates, the job manager unregisters with the job broker. The job manager is the central authorizing agent for the job. Any time a node manager forks a worker, it receives a copy of the job description directly from the job's job manager.

When a node manager forks a new worker, it must take special precautions that the user specified in the job description actually authorized the job to be run. Failure to do so would allow an outsider to gain unauthorized access to a user's account. Furthermore, it is desirable for the macroscheduler's protocols to be secure against unauthorized messages. For these reasons, all of the macroscheduler's protocols are secured with an abstraction on top of UDP called *secure active messages* [30]. This abstraction maintains all of the semantics of the split-phase protocols mentioned earlier but adds a guarantee of the authenticity of messages to the receiver. Unlike a normal UDP message which is sent from one network address to another, a secure active message is sent between "principals." A *principal* is a pair consisting of a network address and a claim as to the identity of the sender. Each secure active message contains user data, the sending principal, and whatever additional data might be required by the underlying authentication protocol, whether that be the standard UNIX `rsh` protocol or a protocol like Kerberos [34]. The security layer is very simplistic, providing only enough functionality to allow the protocols to be secured in a manner independent of the authentication protocol.

The decision to receive the job description directly from the job manager stems from security considerations with protocols like Kerberos, where the job broker and node managers are not trusted with the user's credentials. Only the user in the possession of tickets can be trusted to start a remote process. Since the job manager runs as the desired user, retrieving the job description directly and securely from the job manager assures that the user

has actually authorized running the job.

The task of scheduling jobs on workstations is shared between the job broker and the node managers. The job broker is responsible for ensuring that each job is running on at least one workstation. The node managers then use a distributed, randomized algorithm to divide the workstations evenly among the jobs. Because the node managers are capable of performing scheduling, temporary outages of the job broker do not impede progress in scheduling jobs on the network of workstations. We are currently experimenting with a distributed, randomized macroscheduling algorithm that uses steal rates to estimate worker utilization. Each job should get its fair share of the idle machines, but no job should get more machines than it can efficiently utilize.

## 7 Related work

Cilk-NOW is unique in delivering adaptive and reliable execution for parallel programs on networks of workstations. Traditionally, systems such as PVM [41], TreadMarks [2], and others [11, 16, 23, 29] that are designed to support parallel programs on networks of workstations have not provided adaptive parallelism or fault tolerance. On the other hand, most systems that do provide support for adaptive execution or fault tolerance take a “process-centric” approach. That is, they provide an abstraction of mobile processes and/or an abstraction of reliable processes. As such these systems are very general in their potential application, but they do not provide much support for parallel programs. In contrast, Cilk-NOW does provide support for parallel programs and it does provide adaptive parallelism and fault tolerance; but it does so only for the Cilk parallel programming model. Such specificity allows the Cilk-NOW design to take an end-to-end approach [38] that leverages properties of the Cilk programming model in order to implement adaptive parallelism and fault tolerance simply and efficiently.

Distributed operating systems [17, 36, 43, 46] and remote execution facilities [18, 19, 31, 35, 47] provide services such as remote process execution and, in some cases, process migration. These systems are not intended to be parallel programming environments, though presumably a parallel programming environment could be built atop one of these systems. In fact, Orca [42], which has been built on top of Amoeba, is such a system. These systems are process-centric in that they adapt only by remotely executing and/or migrating processes.

A small number of parallel programming and runtime systems have been built that are adaptively parallel, but unlike Cilk-NOW, none are fault tolerant. Possibly the first adaptively parallel system is the Benevolent Bandit Laboratory (BBL) [22], and Cilk-NOW borrows some of its overall system architecture from BBL. The Pi-

ranha system [24, 27], which is based on the Linda programming model [12], is also adaptively parallel. (In fact, the authors of Piranha appear to have coined the term “adaptively parallel.”) These systems support programming models that are quite different from Cilk’s, but as with the Cilk-NOW design, both leverage properties of their programming model in order to implement adaptive parallelism. A runtime system for the programming language COOL [14] running on symmetric multiprocessors [44] and cache-coherent, distributed, shared-memory machines uses process control to support adaptive parallelism. This system relies on special-purpose operating system and hardware support. In contrast, Cilk-NOW supports adaptive parallelism entirely in user-level software on top of commercial hardware and operating systems. The Spawn system [45] supports concurrent applications with dynamic and adaptive resource management policies based on microeconomic principles. Unlike Cilk-NOW, none of these systems are fault tolerant.

A growing number of systems do provide fault tolerance, but unlike Cilk-NOW, none provide “application” fault tolerance in a high-level parallel programming environment. The Hive [15] distributed operating system provides “system” fault tolerance, meaning that a fault in one component does not bring down the entire system. Hive does not, however, provide “application” fault tolerance, meaning that with Hive, if an application is using a failed component, then the entire application crashes (unless the application itself has taken care to be fault tolerant). Application fault tolerance is provided by the Manetho system [21] via the technique of message logging. The Sam system [39] uses message logging to implement a fault tolerant distributed shared memory. The Sam implementation leverages properties of its shared-memory consistency model in order to avoid logging certain messages. Unlike Cilk-NOW, both Manetho and Sam are process-centric, as they both provide the abstraction of reliable processes, and neither is really a high-level parallel programming environment.

In comparing Cilk-NOW with these other process-centric systems, an interesting question to ask is, why not build the Cilk-NOW runtime system on top of one of these other systems? After all, these systems already implement adaptive and/or fault-tolerant execution. The answer is performance. As we have seen, the overhead of adaptive parallelism and fault tolerance in Cilk-NOW is amortized against the overhead in Cilk’s provably efficient scheduling algorithm. This amortization is only possible because all facets of the design are specialized with high-level knowledge of algorithmic structure in the Cilk programming model.

## 8 Conclusion

The widespread use of NOWs for parallel computation requires a software infrastructure that allows programmers to code in a high-level language that abstracts away the complexity of protocols, scheduling, and resource management. Cilk and Cilk-NOW are part of this developing software infrastructure. In this paper, we have shown how Cilk-NOW's end-to-end design leverages structure in the Cilk programming model to implement adaptive parallelism and fault tolerance simply and efficiently. All overheads are amortized against work-stealing operations, and the number of steals grows with the critical path and not with the work. This result is only possible because Cilk-NOW incorporates Cilk-specific policies at all levels of its design.

The Cilk-NOW runtime system, as described in this paper and as currently implemented, supports the Cilk-2 language which is essentially functional in that it does not have support for a global address space or parallel I/O. More recent incarnations of Cilk for MPPs and SMPs have support for a global address space using "dag-consistent" distributed shared memory [7], and we are currently working on extensions for parallel I/O. With these additions to Cilk, preserving Cilk-NOW's adaptive and fault tolerant execution model remains a challenging open problem. We are currently working on this problem. The dag-consistency model was conceived with adaptive parallelism and fault tolerance in mind, and we are investigating the idea of coupling our current return transactions mechanism with a causal message-logging mechanism [1].

In other current research, we are investigating distributed macroscheduling algorithms. The goal of such an algorithm is to assign idle workstations to Cilk jobs so that each job gets a "fair" share and without requiring that users explicitly state their application's resource needs. It turns out that the parallelism of a Cilk job can be determined continuously and automatically by monitoring the steal rate. We are examining a macroscheduling algorithm in which this information is used in randomized pairwise interactions among processors. The idea is that periodically (and asynchronously) each processor picks another processor in the network at random, and if the two processors are working on different jobs, then one processor may switch to the job of the other. The switching decision is randomized and based only on information about the parallelism and size of the two jobs involved. Early simulation results indicate that such a scheme is very effective [30], and we are currently working on analysis and implementation.

More information about Cilk, including papers, documentation, and software releases, but not including Cilk-NOW software, can be found on the World-Wide Web at

<http://theory.lcs.mit.edu/~cilk>.

## Acknowledgments

We wish to thank the entire Cilk project team at MIT. Led by Professor Charles E. Leiserson, this team includes or has included Matteo Frigo, Michael Halbherr now of the Boston Consulting Group, Chris Joerg now of DEC's Cambridge Research Lab, Bradley Kuszmaul now of Yale University, Howard Lu, Rob Miller now of Carnegie Mellon University, David Park now of McKinsey and Associates, Keith Randall, and Yuli Zhou now of AT&T Labs Research. Of particular note in this group, we wish to extend an extra note of gratitude to David Park, who helped conceive and write the very first Cilk-NOW prototype (back then we called the system "Phish") and especially to Charles Leiserson. From its conception, Charles has supported this project with a generous allocation of resources, both human and machine, and with his enthusiasm, ideas, and sense of humor.

Other current and former members of MIT's Laboratory for Computer Science who we wish to thank include Arvind, Scott Blomquist, Eric Brewer now of the University of California at Berkeley, Frans Kaashoek, Larry Rudolph, and Sivan Toledo now of IBM.

Finally, we wish to thank the anonymous reviewers for their many helpful comments and our "shepherd," Fred Douglass, for his patience and guidance in helping us address the reviewers' comments.

## References

- [1] Lorenzo Alvisi and Keith Marzullo. Message logging: Pessimistic, optimistic, causal and optimal. In *Proceedings of the 15th IEEE International Conference on Distributed Computing Systems*, pages 229–236, Vancouver, Canada, June 1995.
- [2] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.
- [3] Thomas E. Anderson, David E. Culler, and David A. Patterson. A case for NOW (networks of workstations). *IEEE Micro*, 15(1):54–64, February 1995.
- [4] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, New York, 1992.
- [5] Remzi H. Arpaci, Andrea C. Dusseau, Amin M. Vahdat, Lok T. Liu, Thomas E. Anderson, and David A. Patterson. The interaction of parallel and sequential workloads on a network of workstations. In *Proceedings of the 1995 ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, pages 267–278, May 1995.

- [6] Robert D. Blumofe. *Executing Multithreaded Programs Efficiently*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, September 1995.
- [7] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. Dag-consistent distributed shared memory. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS)*, pages 132–141, Honolulu, Hawaii, April 1996.
- [8] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, Santa Barbara, California, July 1995.
- [9] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 356–368, Santa Fe, New Mexico, November 1994.
- [10] Robert D. Blumofe and David S. Park. Scheduling large-scale parallel computations on networks of workstations. In *Proceedings of the Third International Symposium on High Performance Distributed Computing (HPDC)*, pages 96–105, San Francisco, California, August 1994.
- [11] Clemens H. Cap and Volker Strumpfen. Efficient parallel computing in distributed workstation environments. *Parallel Computing*, 19:1221–1234, 1993.
- [12] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.
- [13] Vint Cerf and Robert Kahn. A protocol for packet network intercommunication. *IEEE Transactions on Computers*, 22(5):637–648, May 1974.
- [14] Rohit Chandra, Anoop Gupta, and John L. Hennessy. COOL: An object-based language for parallel programming. *IEEE Computer*, 27(8):13–26, August 1994.
- [15] John Chapin, Mendel Rosenblum, Scott Devine, Tirthankar Lahiri, Dan Teodosiu, and Anoop Gupta. Hive: Fault containment for shared-memory multiprocessors. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 12–25, Copper Mountain Resort, Colorado, December 1995.
- [16] Jeffrey S. Chase, Franz G. Amador, Edward D. Lazowska, Henry M. Levy, and Richard J. Littlefield. The Amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles (SOSP 12)*, pages 147–158, Litchfield Park, Arizona, December 1989.
- [17] David R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.
- [18] Henry Clark and Bruce McMillin. DAWGS—a distributed compute server utilizing idle workstations. *Journal of Parallel and Distributed Computing*, 14(2):175–186, February 1992.
- [19] P. Dasgupta, R. C. Chen, S. Menon, M. P. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R. J. LeBlanc, W. F. Appelbe, J. M. Bernabéu-Aubán, P. W. Hutto, M. Y. A. Khalidi, and C. J. Wilkenloh. The design and implementation of the Clouds distributed operating system. *Computing Systems*, 3(1):11–46, 1990.
- [20] Fred Douglass and John Ousterhout. Transparent process migration: Design alternatives and the sprite implementation. *Software—Practice and Experience*, 21(8):757–785, August 1991.
- [21] Elmootazbellah N. Elnozahy and Willy Zwaenepoel. Manetho: Transparent rollback-recovery with low overhead, limited rollback and fast output commit. *IEEE Transactions on Computers*, C-41(5):526–531, May 1992.
- [22] Robert E. Felderman, Eve M. Schooler, and Leonard Kleinrock. The Benevolent Bandit Laboratory: A testbed for distributed algorithms. *IEEE Journal on Selected Areas in Communications*, 7(2):303–311, February 1989.
- [23] Vincent W. Freeh, David K. Lowenthal, and Gregory R. Andrews. Distributed Filaments: Efficient fine-grain parallelism on a cluster of workstations. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 201–213, Monterey, California, November 1994.
- [24] David Gelernter and David Kaminsky. Supercomputing out of recycled garbage: Preliminary experience with Piranha. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, pages 417–427, Washington, D.C., July 1992.
- [25] Chris Joerg and Bradley C. Kuszmaul. Massively parallel chess. In *Proceedings of the Third DIMACS Parallel Implementation Challenge*, Rutgers University, New Jersey, October 1994. Available as <ftp://theory.lcs.mit.edu/pub/cilk/dimacs94.ps.2>.
- [26] Christopher F. Joerg. *The Cilk System for Parallel Multithreaded Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, January 1996.
- [27] David Louis Kaminsky. *Adaptive Parallelism with Piranha*. PhD thesis, Yale University, May 1994.
- [28] Phillip Krueger and Rohit Chawla. The Stealth distributed scheduler. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 336–343, Arlington, Texas, May 1991.
- [29] Kai Li. IVY: A shared virtual memory system for parallel computing. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 94–101, August 1988.
- [30] Philip Andrew Lisiecki. Macro-level scheduling in the Cilk network of workstations environment. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1996.
- [31] Michael J. Litzkow, Miron Livny, and Matt W. Mutka. Condor—a hunter of idle workstations. In *Proceedings of the 8th International Conference on Distributed Comput-*

- ing Systems, pages 104–111. San Jose, California, June 1988.
- [32] Howard J. Lu. Heterogeneous multithreaded computing. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1995.
  - [33] Robert C. Miller. A type-checking preprocessor for Cilk 2, a multithreaded C language. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1995.
  - [34] Steven P. Miller, B. Clifford Neuman, Jeffrey I. Schiller, and Jeremie H. Saltzer. Kerberos authentication and authorization system. Athena technical plan, M.I.T. Project Athena, October 1988.
  - [35] David A. Nichols. Using idle workstations in a shared computing environment. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles (SOSP 11)*, pages 5–12, Austin, Texas, November 1987.
  - [36] John K. Ousterhout, Andrew R. Cherenson, Frederick Douglass, Michael N. Nelson, and Brent B. Welch. The Sprite network operating system. *IEEE Computer*, 21(2):23–36, February 1988.
  - [37] Vijay S. Pande, Christopher F. Joerg, Alexander Yu Grosberg, and Toyochi Tanaka. Enumerations of the hamiltonian walks on a cubic sublattice. *Journal of Physics A*, 27, 1994.
  - [38] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.
  - [39] Daniel J. Scales and Monica S. Lam. Transparent fault tolerance for parallel applications on networks of workstations. In *Proceedings of the USENIX 1996 Annual Winter Technical Conference*, San Diego, California, January 1996.
  - [40] W. Richard Stevens. *UNIX Network Programming*. Prentice-Hall, Englewood Cliffs, New Jersey, 1990.
  - [41] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.
  - [42] Andrew S. Tanenbaum, Henri E. Bal, and M. Frans Kaashoek. Programming a distributed system using shared objects. In *Proceedings of the Second International Symposium on High Performance Distributed Computing*, pages 5–12, Spokane, Washington, July 1993.
  - [43] Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, Sape J. Mullender, Jack Jansen, and Guido van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, December 1990.
  - [44] Andrew Tucker and Anoop Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles (SOSP 12)*, pages 159–166, Litchfield Park, Arizona, December 1989.
  - [45] Carl A. Waldspurger, Tad Hogg, Bernardo A. Huberman, Jeffrey O. Kephart, and W. Scott Stornetta. Spawn: A distributed computational economy. *IEEE Transactions on Software Engineering*, 18(2):103–117, February 1992.
  - [46] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The LOCUS distributed operating system. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles (SOSP)*, pages 49–70, Bretton Woods, New Hampshire, October 1983.
  - [47] Songnian Zhou, Jingwen Wang, Xiaohu Zheng, and Pierre Delisle. Utopia: A load sharing facility for large, heterogeneous distributed computer systems. *Software—Practice and Experience*, 23(12):1305–1336, December 1993.



# A Distributed Shared Memory Facility for FreeBSD\*

Pedro Souto<sup>†‡</sup> and Eugene W. Stark<sup>§</sup>

Department of Computer Science  
State University of New York at Stony Brook  
Stony Brook, NY 11794 USA

## Abstract

*This paper describes the design and implementation of a distributed shared memory facility we have implemented for the FreeBSD operating system (a descendant of 4.4BSD that runs on the PC architecture). Interesting aspects of the design are: (1) the consistency protocol uses unreliable datagram communication, but is robust with respect to message loss, and in the normal case requires only two datagrams to handle a read fault; (2) the facility provides a simple programming interface that does not require any socket or network programming to use; (3) the facility extends the FreeBSD VM system in a very non-intrusive way.*

## 1 Introduction

A distributed shared memory (DSM) facility permits processes running at separate hosts on a network to share virtual memory in a transparent fashion, as if the processes were actually running on a single processor [LH89]. This is accomplished with the help of the virtual memory (VM) subsystem, which identifies page faults on DSM pages and invokes the DSM subsystem to retrieve data over the network and perform necessary synchronization.

This paper describes the design and implementation of a distributed shared memory system we have built for the FreeBSD 2.1 operating system, a descendant of 4.4BSD that runs on the PC architecture. The following goals were important in shaping the design of our facility:

- A simple client application interface, which would be as close as possible to ordinary memory.

\*Product names used in this publication are used for identification purposes only and may be trademarks of their respective companies or organizations.

<sup>†</sup>Supported by a PRAXIS XXI fellowship, from the Junta Nacional Científica e Tecnológica of Portugal.

<sup>‡</sup>E-mail address: [souto@cs.sunysb.edu](mailto:souto@cs.sunysb.edu)

<sup>§</sup>E-mail address: [stark@cs.sunysb.edu](mailto:stark@cs.sunysb.edu)

- To make the basic DSM read page/write page operations as efficient as possible in the normal case.
- A nice fit with the existing FreeBSD VM system, with minimal changes to existing FreeBSD kernel code, and providing flexibility for experimentation with different consistency protocols.

The programming model presented to client applications centers around the notion of DSM *objects*, which are used in a fashion analogous to the use of memory-mapped files. No network or socket programming is required of an application in order to use the DSM facility. At the kernel level, the notion of DSM object fits together nicely with the VM objects that already exist in the Mach-derived FreeBSD VM subsystem, enabling us to add the DSM facility to FreeBSD in a very non-intrusive fashion.

To achieve efficiency and low communication complexity, we adopted as a basic design decision that unreliable datagram communication (our implementation uses UDP) should be used whenever possible. The protocol we designed, which is a write-invalidate protocol that ensures sequential consistency [Lam79], requires in the normal case only two datagrams (request and reply) to retrieve a copy of a page from a remote host, and a total of  $n + c + 1$  datagrams (or an  $n$ -way multicast plus  $c + 1$  individual datagrams) to obtain write permission on a copy of a page, where  $n$  is the number of hosts interested in the object and  $c$  is the number of hosts that actually hold copies of the page. The protocol is robust in the face of loss or reordering of datagrams, though in this case or in the case of contention for pages, additional messages may be required. Measurements of basic latencies show that our read page fault are less than 3 ms, which is within 1.5 ms of the best published results [BB93] we know of.

The implementation of the DSM facility required

only minimal changes to the existing FreeBSD kernel code: only about 100 lines of additions or modifications were made to previously existing kernel files. The rest of the system is split between about 3000 lines of new kernel code and about 5000 lines for a user-level DSM server program. The user-mode server implements essentially all aspects of the consistency protocol, and interacts with the kernel through a narrow interface, thus allowing easy experimentation with various consistency protocols.

The remainder of the paper describes in more detail some of the more interesting aspects of our system.

## 2 Architectural Overview

### 2.1 Programming Model

Our DSM facility centers around the concept of a *DSM object*, which is a virtual address space that consists of a sequence of shared pages. A process wishing to access a DSM object must first obtain for that object: (1) a UID, which uniquely identifies that object among all other DSM objects in the world, and (2) the network address of a DSM server that knows about that object. A UID is obtained either by requesting the creation of a new DSM object, or else by receiving the UID of an existing DSM object through some communication channel outside the DSM facility. Network addresses are obtained by similar means. Once a process has the UID of a DSM object, and a corresponding server address it requests to *attach* to the object, using a system call provided for this purpose. After attaching to the object, the process uses another system call to *map* pages from the DSM object into its own virtual address space. When a process has finished accessing a DSM object, it asks to *detach* from the object; in response to this request the DSM facility deletes any existing mappings of that object. The attach/map/detach paradigm for DSM objects is analogous to the open/map/close paradigm for memory-mapped files. It is also quite similar to what is provided by the System V *shm* [ATT90] shared memory facility for interprocess communication.

Once a process has mapped DSM pages into its virtual address space, normal memory references to the mapped virtual addresses are used to access data in the DSM object. As usual, such memory references will cause a *page fault* if either the corresponding page is not resident in physical memory, or else the page does not have the appropriate access permissions set. When a page fault occurs for a virtual address that has been mapped to a DSM object, the kernel page fault handler dispatches a request to the DSM subsystem. The DSM subsystem

handles this request, communicating, if necessary, with DSM servers elsewhere in the network either to obtain a copy of the page to be read, or else to synchronize with the other servers to ensure that a write operation can be performed on a page without violating data consistency guarantees. Once the required communication and synchronization has been performed, the DSM subsystem responds to the page fault handler, and the faulting process is allowed to continue.

An important feature of the above programming model is that processes using the DSM feature do not have to contain any code for communication over the network. The only aspect of network programming that shows through the interface is the network address required initially to attach to a DSM object, however, this network address can be treated opaquely, as simply a string of bits that is passed to the kernel as an argument to the *attach* request.

### 2.2 System Structure and Kernel Interfaces

The DSM subsystem has a client/server structure, and contains both kernel and user-level components. The overall organization is depicted in Figure 1. *Clients* are the user-level application processes that make use of the DSM facility. Arbitrarily many clients can run on a single host computer. To support the DSM operations of the clients, a single DSM *server* process runs on each host computer providing DSM service. The DSM server is also a user-level process, though it is a privileged process that makes use of special DSM system calls provided by the kernel. The kernel portion of the DSM subsystem consists of (1) DSM *pager* code, which runs on behalf of a client process as a result of a page fault, (2) *client system calls*, which allow clients to attach, map, and detach DSM objects as described above, and (3) *server system calls*, which provide the DSM server process with the access to the VM system it needs to carry out its function.

As described above, the kernel page fault handler invokes the DSM pager code in response to a page fault by a client process involving a virtual address that has been mapped to a page in a DSM object. The DSM pager does not itself perform any communication or synchronization with remote DSM servers. Instead, it sends a request datagram to the local DSM server indicating the type of service that is required, and then sleeps awaiting a response. The local DSM server receives and handles this request datagram, possibly communicating with DSM servers elsewhere in the network as a result. When the required communication and synchronization has been performed, and any requested DSM

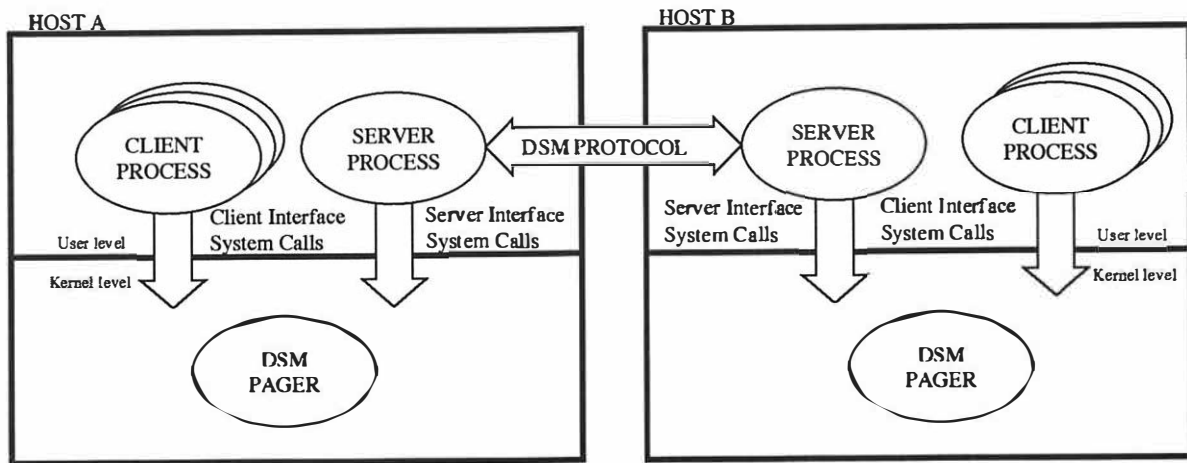


Figure 1: General architecture of the DSM facility.

data is available in local physical memory, the DSM server uses a special system call to awaken the client process sleeping in the DSM pager. The special system call is used so that the DSM server can wake up clients directly in the kernel, instead of requiring every client application to contain code for receiving and interpreting reply datagrams from the server.

To simplify the structure of the DSM server program, no attempt is made by the server to keep track of the status of client operations in progress and to ensure they succeed. Thus, it is possible that a request sent by the local DSM server to a DSM server on a remote host might fail to elicit a response from the remote host; this failure in turn would mean that the local server might never respond to the client that issued the request. In such a situation, the onus is on the client to get things moving again: if the local DSM server fails to awaken the client process after a suitable interval, the client times out from the kernel sleep routine and resubmits the request to the server.

The client system call interface consists of the following system calls (Figure 2a): `dsmcreate`, `dsmattach`, `dsmdetach`, `dsmmap` and `dsmwait`.

The `dsmcreate` call takes as an argument a size in bytes, and causes a new DSM object of that size to be created. The UID of the newly created object is returned. The `dsmattach` call takes as arguments the UID of a DSM object and the network address at which a server who knows about that object can be contacted, and it arranges for the calling process to become attached to the specified DSM object. The return value indicates success or failure. The `dsmdetach` call takes the UID of a DSM object as its single argument, and it causes the calling process to become detached from the specified object. The

`dsmcreate`, `dsmattach`, and `dsmdetach` system calls are not actually executed in the context of the client process. Rather, they cause a request datagram to be sent to the local DSM server, who performs the requested service and returns a response to the client waiting in the kernel.

The client `dsmmap` call is identical to that of the previously existing `mmap` call, used to memory map files, except that `dsmmap` requires the UID of a DSM object to be supplied instead of a file descriptor. In spite of the overlap between `dsmmap` and `mmap`, we chose to keep them separate in the current implementation to avoid modifications to existing code. The `dsmwait` call is used by a client process to avoid expensive busy waiting on DSM data. It takes as arguments the UID of a DSM object and the offset of a particular byte in that object, and it causes the caller to sleep as long as it can be guaranteed that the data byte at that offset has not been changed. As soon as this guarantee can no longer be made (for example, if a remote host obtains write permission on the page containing the particular byte), the server awakes the client, which returns to user mode.

The server system call interface consists of the following system calls (see Figure 2b): `dsmserve`, `dsmcreate`, `dsmdelete`, `dsmrespond`, `dsminvalid`, `dsmwritepage`, `dsmsepage`, and `dsmrecpage`. The `dsmserve` call is used by the DSM server process on startup to identify itself to the kernel, to prevent any other DSM server processes from starting, and to enable access to the remaining server calls. The `dsmcreate` and `dsmdelete` calls are used to inform the kernel of the creation and deletion of a DSM object. The `dsmrespond` call is used by the server to wake up a client process sleeping in the kernel while awaiting DSM service. The DSM server uses

```

int dsmcreate(int size);
int dsattach(int objid, struct sockaddr *addr, int len);
int dsdetach(int objid);
caddr_t dsmap(caddr_t addr, int len, int prot, int flags, int objid, off_t offset);
int dswait(int objid, off_t offset);

```

a) Client system call interface.

```

int dsmservice(int socket);
int dsmcreate(int size);
int dsdelete(int objid);
int dsrespond(int responseid, int result);
int dswritepage(int objid, off_t offset);
int dsminvalid(int objid, off_t offset);
int dsmsendpage(int objid, off_t offset, dsm_data_packet_t dpkt,
                struct sockaddr *addr, int len);
int dsrecvpage(int objid, off_t offset);

```

b) Server system call interface.

Figure 2: System call interface.

the `dsfwritepage` call to tell the kernel that it is safe to write enable a particular page of DSM data. The `dsminvalid` call causes the kernel to invalidate any copies it may have of a particular page of DSM data, so that subsequent attempts by clients to access these pages will fault. Finally, the `dsmsendpage` and `dsrecvpage` are called by the server to send and receive a page of DSM data over the network. System calls are provided for this in order to enable the transmission and reception of DSM data directly from or to the appropriate page of physical memory, so that the user-level server process never touches the actual DSM data. Without these calls, sending a page of DSM data to a remote host would be a much more costly operation involving the copying of data from the kernel to the server process' address space, then copying the data back into the kernel for transmission over the network, followed by the reverse sequence at the destination host.

Note that although the system call interface provides several logically separate system calls, in fact the implementation uses only one actual system entry, `dsmsys()`, which dispatches on its first argument to invoke the appropriate function. This scheme saves system call numbers and is similar to the system call interface of System V shared memory.

In the first version of our system, the DSM server executed completely in the kernel, similar to what occurs in the NFS network file system. This was done for efficiency reasons, and because at the outset we did not have a clear picture of what sort of kernel interface would be required for a user-level server. Unfortunately, the complexity of the

server data structures and storage management issues were such that it became too difficult to completely debug a kernel-mode server. In fact, one of the most difficult aspects of the server implementation was implementing a suitable reference counting scheme for DSM data structures, so that DSM resources would be reclaimed automatically when no processes were using them any more. To aid in debugging, we decided to reimplement the server as a user-level process, which communicates with the kernel through the narrow, system call interface just described. This interface is largely independent of the details of the consistency protocol, a feature we have found very useful while refining and debugging our particular protocol.

### 3 Details of the Kernel DSM Subsystem

An important design goal for our DSM facility was to have it mesh nicely with the structure of the FreeBSD VM system, and to require minimal modifications to existing code. We feel we were reasonably successful at meeting this goal; in the rest of this section we describe in more detail some of the more interesting aspects of the design.

The FreeBSD virtual memory system is based on that of 4.4BSD [MBKQ96], which in turn is derived from that of Mach (Figure 3). A fundamental concept in the Mach VM system [Tev87] is the concept of a VM *object*, which consists essentially of a placeholder for a sequence of physical pages, together with an associated *pager*, which is a set of functions that can be invoked to retrieve data from a backing store,

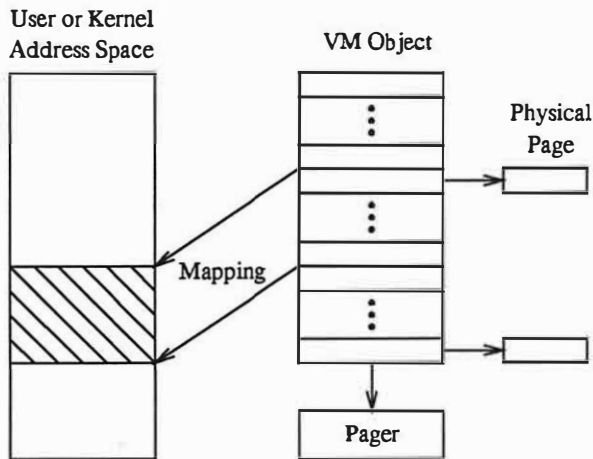


Figure 3: Simplified view of the Mach VM architecture.

such as a swap area, a file, or a hardware device. A process obtains access to the data in a VM object by mapping some or all of its pages into its address space. Typically a process has only a few mappings of VM objects at a time, but actually there is no limit on the number of mappings or the number of VM objects whose pages are mapped.

In the FreeBSD VM system, the allocation of physical memory pages for a VM object is decoupled from the mapping of the object into a process' address space. Physical pages only need to be allocated in a VM object when an attempt is actually made by a process to access data at an offset in a VM object for which no page has yet been allocated. Such an attempt produces a page fault, and the page fault handler not only allocates a physical page for that data, but also invokes the pager to retrieve the data from backing store. The stock FreeBSD VM system supports three different types of pagers: a *swap pager*, which manages the swap area, a *vnod pager*, which is used to page data to and from disk files, and a *device pager*, which is used to page data directly to a hardware device. Additional types of pagers are easily defined.

Our DSM facility was designed to take advantage of the existing FreeBSD VM subsystem. Each DSM object contains an underlying VM object. Mapping a DSM object into a process' address space amounts to simply mapping the underlying VM object. To support the fetching of data over the network, we introduced a new type of pager, called a *DSM pager*. When a fault occurs on a memory address that is mapped to a page in a DSM object, the page fault handler invokes the DSM pager. The DSM pager determines the status of that page by checking data

structures maintained by the DSM subsystem, and then requests the local DSM server to perform any communication or synchronization required to bring a copy of the data into local physical memory, and to write enable it, if necessary.

At the rather coarse level of detail of the description so far, the interaction of the DSM facility with the page fault handler seems quite simple. However there are some technical issues that make things a bit more complex than they seem at first. First of all, the stock FreeBSD page fault handler is an extremely complex routine involving many subtle synchronization issues. In order to avoid a difficult debugging task, and to make it easier to track future releases of FreeBSD, we wanted to modify as little of the page fault handler as possible. Some modification to the page fault handler was necessary, because whereas the DSM pager needed to be informed as to whether a read fault or write fault was being handled, the pager interface in FreeBSD did not contain any provision for passing this information to the pager from the page fault handler.

A second technical issue was that the DSM subsystem had to be responsive to requests from the pageout daemon to clean pages of physical memory. The obvious thing for the DSM pager to do when asked to clean a page would be to write it to the swap area. However, I/O to the swap area has to be asynchronous to avoid blocking the pageout daemon, and since there was already an existing swap pager that contained the complicated code necessary to perform this asynchronous I/O, we wanted to make use of it if possible.

A third issue was how the user-level DSM server could arrange for DSM data stored at the local host to be transmitted over the network, even if this data happens to currently reside in the swap area.

To understand how we dealt with the above technical issues, it is necessary for us to describe one more feature of the FreeBSD/Mach VM system. In the FreeBSD VM system, VM objects can be linked together in so-called "shadow chains," which have a special significance to the page fault handler. When a page fault occurs for a page mapped to the first object in a chain, the page fault handler first consults the associated pager (if any) for that object, to try to handle the fault. If the pager for the first object fails to handle the fault, then the page fault handler tries the second object in the chain, and so on. Thus, each object in such a chain serves as a "backing object" for the preceding object, in the sense that if a page is not found in the preceding object, an attempt is made to obtain the page from the next object. In the original literature [Tev87]

describing this scheme, each object in a chain is said to be a “shadow” of the next object. However, this terminology has turned out to be very confusing, so we prefer to use the “backing object” terminology instead.

A major purpose of the chains of VM objects in FreeBSD is to support “copy-on-write” for efficient forking. However, for the DSM facility we use these chains in a different way (Figure 4), which we now describe. We have already mentioned that each DSM object has an underlying VM object, which is the actual target of mapping operations by a process. This underlying object has an associated DSM pager, which encapsulates knowledge of how to obtain pages over the network and how to synchronize with the DSM subsystem at remote hosts. In addition, when a DSM object is created, we also create a second VM object that serves as a backing object for the first, so that underlying a DSM object is always a chain of two VM objects. The pager associated with the backing object is not a DSM pager, but rather a swap pager.

When a page fault occurs for an address mapped into a DSM object, the normal operation of the page fault handler is to check the first of the two underlying VM objects to try to obtain the page. When the DSM pager associated with the first object is invoked, it determines: (1) that no copy of the page is available at the local host, or (2) that a copy of the page is locally available, but it might be paged out locally to the swap area. In case (1), the DSM pager sends a datagram to the local DSM server requesting the transfer of a copy of the page to the local host, and it sleeps awaiting the arrival of the page. In case (2), the DSM pager returns a failure indication to the page fault handler, which then moves to the backing object. The page fault handler either finds the requested data already in physical memory mapped from the backing object, or else invokes the swap pager associated with the backing object to bring the data in from the swap area.

The utility of the two-element chain underlying a DSM object becomes evident when one considers how to implement the “pageout” operation of the DSM pager. Rather than having the DSM pager perform a complicated algorithm for asynchronous I/O to the swap area, in response to a “clean” request from the pageout daemon, the DSM pager simply copies the data from the first object in the chain to the corresponding position in the second object. This doesn’t immediately free up physical memory, but if there is a high demand for memory, the pageout daemon will eventually ask the swap pager associated with the second object in the chain to clean

its page, in which case the data will be written to the swap area using the standard pageout code.

Thus, the two-element chain of VM objects underlying each DSM object permits the DSM system easy access to the normal swap area, with hardly any additional code required. This organization also pays off when the DSM server wishes to transmit to a remote host DSM data that has been paged out locally. As discussed above, transmission of DSM data is accomplished by the special `dsmsendpage` server system call, to minimize the number of copies of a page’s data when sending that page to a remote server. This system call simply maps the page of the *first* object in the two-object chain into the kernel address space, and then copies data from that page to the network subsystem. If the page happens not to be resident in physical memory, a page fault will occur, and since the DSM pager does not have the page, the page fault handler will follow the object chain and find the page in the backing object.

Similarly, the reception of DSM data makes use of the special `dsmrecvpage` server system call, to minimize the number of copies of DSM data during reception. This system call temporarily maps the physical page, which was allocated when the page fault first occurred, into the kernel address space and copies the data from the network subsystem to that page.

To support the scheme described above, the kernel needs to have a certain amount of information about the state of the DSM subsystem. In particular, the kernel keeps a data structure for each DSM object that points to the associated VM objects, and keeps track of the location and status (available locally/available remotely, resident/nonresident, write enabled/write protected) of each page in the object. It also keeps a list of pending DSM requests from clients, so that the proper client process can be identified and awakened when the DSM server responds to such a request. The kernel does not need to know anything about the particulars of the DSM protocol or about remote sites participating in the DSM protocol; this is entirely the responsibility of the user-level DSM server process.

## 4 DSM Protocol

This section describes the DSM protocol executed by our user-level DSM server processes. Essentially, there are two related protocols: (1) a *membership* protocol, which keeps track of hosts that are currently interested in accessing a DSM object, and (2) the *consistency* protocol, which is executed by hosts wishing to read or write pages in DSM objects. The consistency protocol is executed frequently: every

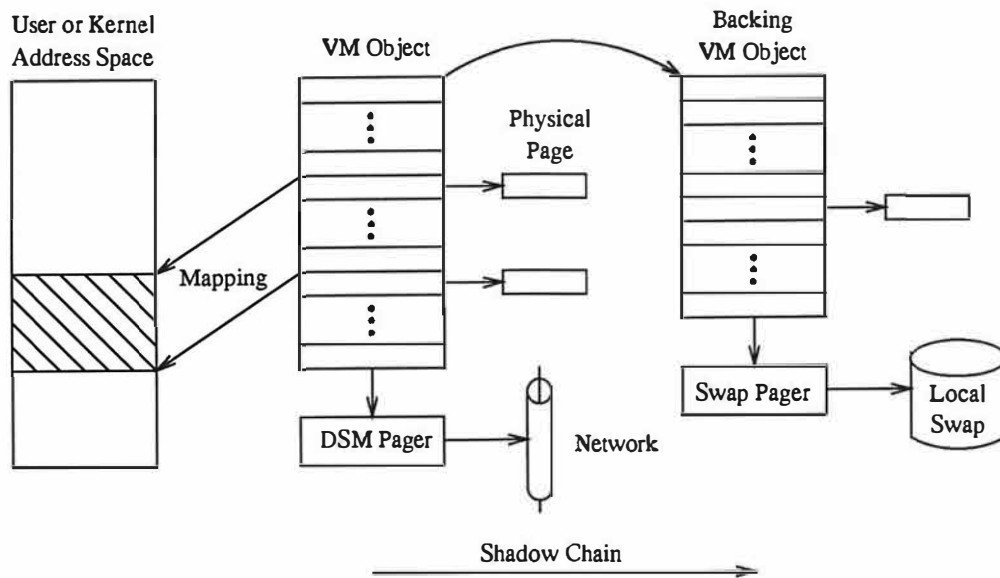


Figure 4: Use of object shadow chains by the DSM facility.

time a host requires an up-to-date copy of a DSM page or needs to obtain write permission on a page. The membership protocol is executed less frequently, but it must be reliable in the sense that the correctness of the consistency protocol depends on the accuracy of the information maintained by the membership protocol. For efficiency, the frequently executed consistency protocol uses unreliable datagrams in all situations but one. On the other hand, the less-frequently executed membership protocol uses reliable datagrams for all communications, where by reliable we mean that a timeout/retransmit scheme is used to guarantee delivery, and ordering of datagrams between each pair of hosts is maintained using a sequence numbering scheme.

To explain the protocol, we first need to introduce some preliminary concepts and terminology. When a DSM object is first created, the only host that knows about that object is the host at which the object is created. This host is called the object *manager*, and it plays a special role in some aspects of the DSM protocol. Application processes at other hosts become informed about the existence of a DSM object by receiving its UID via some form of communication outside the DSM system. The first time an application process at a host tries to attach to the DSM object, the DSM server at that host applies to the manager of the object for *membership* in the object; that is, it asks to be added to the list of all hosts that are currently interested in that object. When the manager grants membership to a new member, it informs all previous members about the new member, and it informs the new member of the current

membership list, so that each member of a DSM object knows at all times who all the other members are. Members of an object can resign their membership at any time; in this case a message is sent to the manager, who informs the remaining members about the change.

The consistency protocol belongs to the class of protocols that Li [LH89] calls “dynamic distributed manager algorithms with page invalidation.” Just as each DSM object has a manager, each page within a DSM object has an *owner*. However, unlike the object manager, which is fixed at the time the object is created and never changes, the owner of a page changes during execution. Initially, the manager of an object owns all pages in the object. Each time a host receives write permission on a page in an object, it becomes the owner of that page. The owner of a page has the responsibility of safeguarding the data in a page, until it has determined which host will be the next owner and has successfully transferred the data to that host.

Each host that is a member of a DSM object maintains the following state information for the object as a whole: (1) the number of local clients attached to this object; (2) the identity of the object manager; (3) the object UID; (4) the current membership list for the object; (5) the size of the object, in bytes and pages.

In addition, for each page in the object, the following state is maintained: (1) an “owner hint” indicating who the current owner of that page might be; (2) a “version hint” indicating the current version number of the page; (3) a “copies hint” indi-

cating how many copies there might be of the page; (4) a flag indicating whether this host has a copy of the page; (5) a flag indicating whether this host has write permission on the page.

The purpose of the owner hint for a page is to try to route requests for a page quickly to the DSM server that has the most recent data for that page. This hint may become stale, but the protocol guarantees that the host mentioned in the hint is always closer to the actual owner than the host holding the hint. Furthermore, the owner hint is always accurate if the host currently has a copy of the page. The version hint is used to filter out datagrams received out of order, which, if processed, might lead the system to an inconsistent state. The copies hint is an estimate of the number of copies of the page that exist. This estimate is conservative in the sense that the estimate held by the owner of a page is always at least as large as the actual number of copies in existence.

There are two basic operations of the DSM protocol: READ (reading a page) and (WRITE) writing a page. We now discuss these operations in some detail.

#### 4.1 Reading a Page

Figures 5a) to 5c) show three interesting cases of a READ operation. Figure 5a) shows a simple READ, in which the host wishing to obtain a copy of a page has an accurate owner hint, and no messages are lost. In this case, only two messages are required: a READ message from the requesting host to the owner, and a DATA reply from the owner. This is the situation we expect to occur most frequently in actual execution. Note that because all messages exchanged in the consistency protocol use unreliable datagrams, there are no hidden acknowledgments or other messages, and so exactly two messages are required to read a page in this situation.

A slightly more complicated case is when the owner hint held by the requesting host is stale. In this case, the host that receives the READ message uses its own hint to forward the message toward the actual owner, as depicted in Figure 5b). The DATA reply goes directly to the requesting host, who updates its owner hint upon receipt. Since owner hints are updated whenever a host receives new information about the owner of a page, we expect that READ messages will generally be forwarded only a few hops.

Figure 5c) shows a scenario in which a DATA message is lost. In this case, a timeout at the requesting host (by a process sleeping in the DSM pager code) triggers the retransmission of the READ message.

This mechanism may lead to the reception of multiple DATA messages containing the same data. To detect this situation, DATA messages include the current version number of the page, and a host receiving a DATA message discards the message if the version is either older than the most recent version of which the host is aware, or the same as the version of any currently held copy. To provide quick error recovery, but to avoid flooding the system with retransmitted READ messages in case of heavy load, we use an exponential backoff scheme with an upper bound to increase the timeout value in case the READ message has to be retransmitted several times.

#### 4.2 Writing a Page

We use a write-invalidate strategy to ensure sequential consistency. That is, before the DSM server at a host allows a client process to modify a page, it invalidates all copies of that page that exist at remote hosts.

In order for a host to initiate a WRITE operation, it is first required to have a copy of the current version of the page. If it wishes to perform a WRITE, but it does not have a current copy, it first executes a READ operation to obtain a copy as described above. There are two reasons for requiring a host wishing to write to have a current copy: (1) it ensures that the host knows the current owner of the page, and (2) it ensures that subsequent message loss during the WRITE operation cannot cause the loss of the data in the page.

Our protocol for writing a page has some uncommon features not usually found in protocols of this type. First, the owner of a page keeps track only of the number of copies of that page, rather than the actual identities of the hosts that have those copies. Second, whereas in all the similar protocols that we know of the write part of the protocol has two distinct operations: the ownership transfer operation and the remote copies invalidation operation, in our protocol the transfer of ownership and the invalidation are combined into a single operation.

Figure 6a) shows what we expect to be the most common case of the WRITE operation, in which there are only a few copies of the page, the owner's copies hint is accurate, and only one host is attempting to write the page. In this situation, the host wishing to write multicasts a WRITE message to every host in the current membership list for the DSM object containing the page to be written. When a member receives the WRITE message, it updates its owner hint to point to the host that issued the WRITE message, and then, if and only if it has a copy of the page, it invalidates that

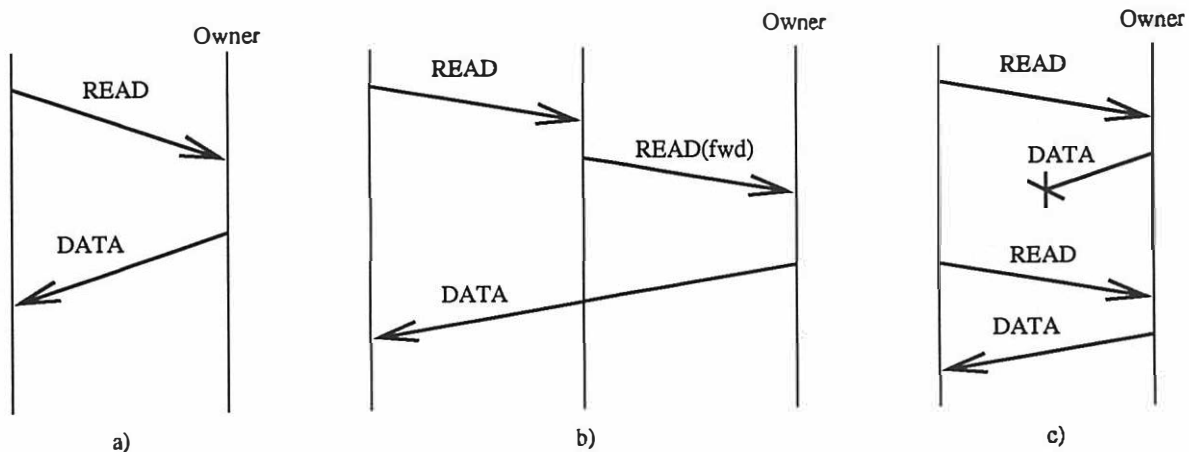


Figure 5: Read operation of the DSM protocol.

copy and responds with a WRITEOK message. The WRITEOK message sent by the current owner of the page implicitly carries with it a transfer of ownership of that page to the requesting host, who, upon receipt of such a message, becomes the new owner, and issues an acknowledgment to the previous owner to release it from any further ownership responsibilities. The WRITEOK message sent by the previous owner also includes a copies hint, which is then used by the new owner to determine when all the remote copies of the page have been invalidated. Specifically, the new owner knows that all copies have been invalidated when the number of WRITEOK messages it has received is equal to the copies hint it received in the WRITEOK message from the previous owner.

As it is absolutely essential that there be no ambiguity about whether ownership has been transferred, the previous owner must wait for its WRITEOK message to be acknowledged, retransmitting the WRITEOK if necessary, before continuing with any other activity regarding this page. The WRITEOK from the previous owner to the new owner is the only reliable datagram used in the consistency protocol. However, observe that the acknowledgement message used by the reliable datagram service is not in the critical path, as the new owner does not have to wait for the acknowledgement to reach the previous owner.

Figure 6b) illustrates what happens in the case of write contention; that is, when two or more hosts try to write the same DSM page at the same time. When the owner of the page processes the first WRITE message for that page, it invalidates its copy of the page and replies with a WRITEOK. If it later sees a WRITE message from some other host for the same version of the page, it simply discards the later

WRITE message. This ensures that only one host will become the owner of the next version of the page, and consequently at most one host will be granted the right to write that page.

A host trying to write a page also discards any WRITE messages it receives. This is necessary, because the only alternative would be for the host to invalidate its copy of the page, but then the page would be lost if the host should happen to be granted ownership by the previous owner. Thus, without any special provisions, a deadlock could result when two hosts try to write the same page and each steadfastly refuses to invalidate and send a WRITEOK to the other. To handle this situation, a host starts a timer when it first multicasts a WRITE message. If, by the time the timer has expired, it has received ownership of the page but has not received enough WRITEOK replies, it multicasts a PURGE message to the membership list. Upon receiving a PURGE message, every member is obligated to invalidate any copy it holds, to update its owner hint to point to the sender, and to reply to the sender with a WRITEOK message. If insufficient WRITEOK messages are received after a suitable period, the owner of the page multicasts another PURGE message. This scenario repeats until the owner is sure that all pages have been invalidated.

To be sure that all copies of a page are actually invalidated, the host issuing a PURGE message has to be able to distinguish WRITEOK messages sent in response to the initial WRITE message, and also in response to subsequent PURGE messages. For this purpose, the host maintains a "phase counter," which is an integer variable that is incremented every time the server multicasts a new set of WRITE or PURGE messages. Each such message includes the value of the phase counter, which the recipient

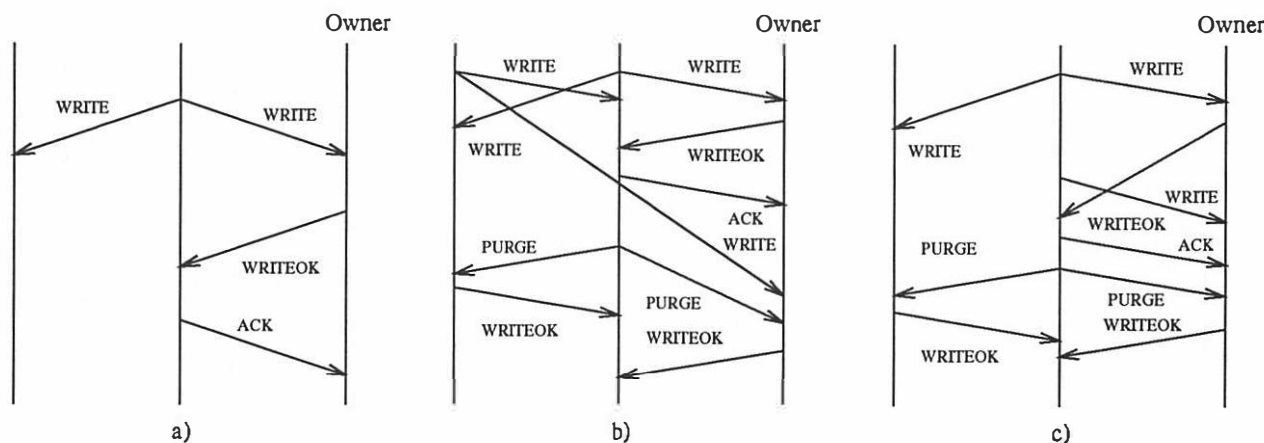


Figure 6: Write operation of the DSM protocol.

echoes back in the WRITEOK response.

Another requirement for correctness of the protocol is that it should not lead to cycles in a page's owner chain. To satisfy that requirement, once a PURGE message has been processed for a version of a page, a host must not again update its owner hint for that page in response to a WRITE message for the same version of that page. Actually, our protocol uses the following stronger policy concerning the updates of owner hints: after processing a WRITE or PURGE message for a version of a page, a host will not again update its owner hint for that page in response to a WRITE message for the same version of that page. However, even if a host has previously received a WRITE message for a version of a page, it *will* still update its owner hint for that page in response to a PURGE message for a version of that page, as long as it is not aware of a more recent version of the page than that specified in the PURGE message.

As mentioned previously, the copies hint held by the owner might not be accurate. For example, due to a slow network or a slow server, a host might retransmit a READ message before it receives the DATA response sent by the owner of the page in response to the original READ message. Since the owner does not keep track of the identity of hosts to which it sent copies of the page, it has no choice upon receiving a retransmitted READ message but to send another DATA message and increment its copies hint to maintain a conservative estimate. This leads to a possibility that a subsequent WRITE operation will deadlock, due to the fact that it will be impossible to obtain enough WRITEOK messages. Eventually, the host wanting to write will time out as described above. If the host has become the owner by the time the timeout occurs, it will multicast a

PURGE message as already described. However, there exists the possibility, due to a slow network or slow response from the previous owner, that by the time the timeout occurs, the host wanting to write has not yet become the owner. In this case, that host simply retransmits the WRITE message to the owner of the page, whose identity it knows because the fact that it has a copy of the page means that its owner hint is accurate. Figure 6c) illustrates such a scenario, in which the previous owner has an inaccurate copies hint and the WRITEOK it sends is slow in arriving at the host performing the WRITE. When the retransmitted WRITE message arrives at the previous owner, it discards it because the WRITEOK has already been sent. Eventually, the host performing the WRITE will time out again. By this time, however, it has received the WRITEOK from the previous owner, and thus will proceed to a PURGE operation as above.

In summary, the basic write protocol just described consists of a sequence of phases: an initial WRITE phase, then zero or more phases in which the WRITE message is retransmitted to the owner of the page, then zero or more PURGE phases.

Message loss has basically the same effect for write operations as does uncertainty about the number of copies of a page: the server does not receive enough WRITEOK messages and consequently cannot be sure whether its copy of the page is the only one in the system. Thus, the mechanisms used to handle inaccuracy of the copy hint also handle the loss of messages.

In order to reduce the write time, we use one optimization that is worth mentioning. As shown in Figure 6c), a very slow network or host can trigger a new WRITE/PURGE phase, due to the late arrival of a WRITEOK message at the host perform-

ing the WRITE/PURGE operation. If the situation persists, the host performing the WRITE/PURGE operation will never receive enough WRITEOK messages for a phase before its timer for that phase expires. To prevent such a scenario, a host performing a WRITE operation keeps track of the WRITEOK messages received for a fixed number of previous WRITE/PURGE phases, and it accepts WRITEOK messages sent in the scope of any such phase, even though it might have initiated several new phases.

## 5 Experimental Results and Discussion

We have run reasonably rigorous tests of the consistency protocol using some simple exerciser programs, and we have measured some basic performance parameters. In this section, we describe the results of these tests. Important testing that we have not yet done is to use the system for a realistic application.

In order to evaluate the performance of our implementation, we performed two sets of experiments: one to determine the basic costs of handling read and write faults on DSM pages, and another to assess the scalability of our protocol. In these experiments we used PC's each with either Pentium 75 MHz or Pentium 100 MHz microprocessors, with 256 Kbyte "write back" second level cache and 16 Mbyte of main memory. These machines are interconnected by an 100 Mbps Ethernet via a SMC EtherPower 10/100 adapter, which sits in the PCI bus and supports DMA.

To determine the costs of both read and write faults, we ran a simple "ping-pong" exerciser program in which two clients running on different hosts alternately write a DSM page in such a way that the page bounces back-and-forth between the two, and there is no write contention. Table 1 shows both the minimum values and the average values over 100,000 operations, for both the read and write times, measured at the kernel level on machines with 75 MHz processors. The read times give the actual time taken to handle a read fault by a client process. The write times include only the time taken to execute the invalidation protocol – in general, handling a write fault may also require an initial read operation to obtain a local copy of the page before beginning invalidation.

These results are encouraging, they are within 1.5 ms of the best published values that we know of [BB93], which were measured on a kernel implementation using a network subsystem specially designed for performance that accesses the network interface directly, bypassing the UDP and other communica-

tions software layers. Note however, that the values presented in that paper were measured using IBM RISC System/6000 Model 530, which run with a clock frequency of 25 MHZ, interconnected by point-to-point 220 Mbps optical fiber network.

It is worth mentioning that a "ping" (ICMP echo request/response) between the machines we used in our experiments takes on average 0.44 ms, and the messages exchanged in a ping neither traverse the UDP layer nor are they processed at user level. We believe that improved performance would result due to reduced IPC costs, if the DSM server were moved into the kernel.

Table 2 shows the breakdown of both the read and write faults times (again, these are averages over 100,000 operations). The IPC times, RD\_IPC and WR\_IPC, measure the time from the moment the DSM pager sends a request to the local DSM server to the time that server starts processing that request. Thus it includes the time to send a message to the local host, the time needed for a context switch, the time to receive the message from a socket, the time spent by that message in a queue of messages in the DSM server, and the time required to do some preprocessing of the message. For this experiment, the message queue at the DSM server is empty, so the message is processed immediately. The DSM server times, RD\_SRV and WR\_SRV, are the times taken by the server to satisfy the DSM pager request; that is, the times taken to get a page from a remote server or to invalidate copies of the page in remote servers.

To assess the scalability of our algorithms we measured the time to invalidate the remote copies of a page under conditions of no contention and when the number of copies of the page is equal to the number of members of the object. As one would hope, the invalidation time depends roughly linearly on the number of remote copies: our measurements showed a constant overhead of 1.3ms, plus an additional 0.4ms for each copy to be invalidated, over the range of 2 to 11 remote copies. Note that our current implementation sends out WRITE messages sequentially. We expect that using a multicast facility for this would decrease the per-copy overhead.

We also ran some experiments under conditions of high write contention. These experiments revealed two potential problems with our current protocol. First, the slower machines (the 75MHz processors) tended to starve in favor of the faster machines (the 100MHz processors). The second problem, which was exacerbated by the first, is that the simple timeout/retransmit policy with exponential backoff we currently use to handle message loss and deadlock

|   | Read | Write |
|---|------|-------|
| minimum time (ms)                       | 2.7  | 2     |
| average time (over 100,000 values) (ms) | 2.9  | 2.2   |

Table 1: Basic read and write times measured at kernel level.

|                   | RD_IPC | RD_SRV | WR_IPC | WR_SRV |
|-------------------|--------|--------|--------|--------|
| average time (ms) | 0.58   | 2.1    | 0.57   | 1.4    |

Table 2: Breakdown of the read and write times shown in Table 1.

situations did not adapt well to varying loads, resulting in a large number of retransmitted messages under conditions of high contention. We are considering ways of improving the timeout heuristics, and of modifying the protocol to alleviate the starvation problem.

A basic aspect of our design that we did not evaluate experimentally was our decision to use UDP rather than TCP for the consistency protocol. We continue to feel that any simplifications in the protocol that might be afforded by the use of TCP as an underlying reliable communications protocol would be more than offset by the overhead of additional acknowledgements, the loss of control over the retransmission policy, and the need for an additional software layer to re-implement a message-based communication model on top of the stream-based TCP protocol. In addition, the use of TCP would not allow us to take advantage of multicast support provided by IP. In spite of the above, to validate our belief in the superiority of datagrams over streams as an underlying protocol, it would probably be worthwhile to perform some experiments in which we compare the performance of the UDP-based version of our protocol with a reasonably similar TCP-based version.

Another interesting question concerns the impact on paging performance of our scheme for “pageout” of DSM pages by copying the data to the second object in the shadow chain. It is possible that the “second chance” this scheme gives to pages containing DSM data could have unforeseen interactions with the pre-existing page replacement policy. To examine these questions, we would have to test our system with a realistic application, under conditions that would cause heavy pageout to the disk. We have not yet performed such tests.

## 6 Related Work

Research in the area of DSM systems has been very intense and there is an extensive literature [Esk96]. We compare our system to other software DSM implementations that support a sequen-

tial consistency model. The main points that we feel distinguish our facility are: (1) The consistency protocol is a lightweight, distributed protocol, which uses unreliable datagrams, but which is robust with respect to message loss, reordering, or duplication. (2) The facility is for a version of Unix (FreeBSD 2.1) for which source code is readily available and which runs on commodity hardware. (3) The clean interface between the user-level server and the kernel should facilitate experimentation with a variety of DSM protocols.

Most of the first implementations of software DSM systems, including that of IVY [Li88], the DSM system for Clouds [RK89] and Mirage [FP89], were implemented in operating systems different from Unix and the consistency protocols used assumed reliable communications. Methner [MF89] is the exception among early implementations. It is a kernel level implementation of DSM for SunOS 4.0. Although it uses UDP, it relies on HW support for error correction. A more recent implementation of Mirage [FHJ94], although for the AIX operating system, uses reliable communication services. Furthermore, every page request has to be sent to the page’s manager, which sends it to the current owner of the page.

The DSM systems described in [FBS89] and [AAO92] take advantage of the VM external pager interface provided by Mach and CHORUS micro-kernels, respectively. The consistency protocol of Mach’s DSM uses only a point-to-point reliable communication service, in contrast to ours which uses multicast and unreliable communication services. Chorus’s DSM uses one of Li’s dynamic manager distributed algorithms with page invalidation [LH89] but the authors do not specify which and their description of the protocol is rather incomplete. In addition, they do not provide details with respect to the kind of communication services used for IPC. As does our DSM system, Chorus’ DSM supports paging out pages to disk, but, in contrast to our system, paging out is handled by the object manager.

Both DVSM6K [BB93], a DSM system developed

for AIX v3, and the DSM system developed for the TOPSY multicomputer [SWS92] have an architecture very similar to that of our system. However, the latter was designed for a distributed memory multiprocessor system using a multiprocessor operating system, and DVSM6K assumes that the communication system provides reliable communication, *i.e.* in-order delivery, no message loss and no data corruption.

## 7 Conclusion and Future Work

In this paper we described a DSM facility, supporting a sequential consistency model, for FreeBSD, a freely and widely available version of Unix. We believe that this facility meets most of our design goals. The consistency protocol is a lightweight protocol that uses only UDP/IP, but is nevertheless tolerant to both message reordering and message loss. We were able to define a very simple client application interface based on the Unix `mmap()` interface. One of the most successful aspects of our design is its smooth integration into the VM subsystem of FreeBSD, which required very little in the way of modifications to existing code. We believe that it should be possible, with minimal effort, to port this code to other Unix systems, such as OSF/1, with Mach-based VM subsystems.

Besides improving the performance of our system in ways that have already been discussed, we are interested in using the facility for real applications. We are especially interested in the idea of using DSM as a tool for programming distributed applications, rather than for concurrent computation, which has been the focus of most DSM research.

## 8 System Availability

We are making our code available to anyone interested under a Berkeley-style copyright and license. The code may be obtained via the URL: <http://www.cs.sunysb.edu/~stark/>, or by mailing to one of the authors.

## 9 Acknowledgements

We wish to thank Professor Tzi-cker Chiueh for making his laboratory facilities available to us, as well as the other members of the Experimental Computer Systems Laboratory for their generous cooperation in the sharing of these facilities.

## References

[AAO92] V. Abrosimov, F. Armand, and M.I. Ortega. A Distributed Consistency Server for the CHORUS System. In *Proc. of the Symposium on Experiences with Distributed and Multiprocessor Systems*

(*SEDMS III*), pages 129–148. USENIX, March 1992.

- [ATT90] ATT. *UNIX SYSTEM V Release 4 - Programmers Guide: System Services and Application Packaging Tools*. Unix Press, 1990.
- [BB93] Marion L. Blount and Maria Butrico. DSVM6K: Distributed Shared Virtual Memory on the RISC System/6000. In *Proc. of the 38th IEEE International Computer Conference (COMPCOM Spring 93)*, pages 491–500. IEEE, February 1993.
- [Esk96] M. Rasit Eskicioglu. A Comprehensive Bibliography of Distributed Shared Memory. *Operating Systems Review*, 30(1):71–96, January 1996.
- [FBS89] A. Forin, J. Barrera, and R. Sanzi. The Shared Memory Server. In *Proc. of the Winter 1989 USENIX Conference*, pages 229–243. USENIX, January 1989.
- [FHJ94] B. D. Fleisch, R.L. Hyde, and N. C. Juul. MIRAGE+: A Kernel Implementation of Distributed Shared Memory on a Network of Personal Computers. *Software - Practice and Experience*, 10(24):887–909, October 1994.
- [FP89] B. D. Fleisch and G. J. Popek. Mirage: A Coherent Distributed Shared Memory Design. In *Proc. of 12th ACM Symposium on Operating Systems Principles (SOSP'89)*, pages 211–223, December 1989.
- [Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C28(9):690–691, September 1979.
- [LH89] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.
- [Li88] Kai Li. IVY: A shared virtual memory system for parallel computing. In *Proc. of the 1988 International Conference on Parallel Processing*, pages 94–101, August 1988.

- [MBKQ96] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison Wesley, 1996.
- [MF89] Ronald G. Minnich and David J. Farber. The Methex System: Distributed Shared Memory for SunOS 4.0. In *Proc. of the Summer 1989 USENIX Conference*, pages 51–60. USENIX, June 1989.
- [RK89] U. Ramachandran and M. Y. A. Khalidi. An Implementation of Distributed Shared Memory. In *Proc. of the Workshop on Experiences with Distributed and Multiprocessor Systems*, pages 21–38. USENIX, October 1989.
- [SWS92] T. Stiemerling, T. Wilkinson, and A. Saulsbury. Implementing DVSM on the TOPSY Multicomputer. In *Proc. of the Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS III)*, pages 263–279. USENIX, March 1992.
- [Tev87] Avadis Tevanian. *Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments*. PhD thesis, Department of Computer Science, Carnegie Mellon University, December 1987.

# Cdt: A General and Efficient Container Data Type Library

Kiem-Phong Vo

AT&T Labs  
600 Mountain Ave  
Murray Hill, NJ 07974  
kpv@research.att.com

## Abstract

*Cdt* is a container data type library that provides a uniform set of operations to manage dictionaries based on the common storage methods: *list*, *stack*, *queue*, *set* and *ordered set*. It is implemented on top of linked lists, hash tables and splay trees. Applications can dynamically change both object description and storage methods so that abstract operations can be exactly matched with run-time requirements to optimize performance. This paper briefly overviews *Cdt* and presents a performance study comparing it to other popular container data type packages.

## 1 Introduction

A data structure to store objects is a container data type and an instance of it is a container or a *dictionary*. Common examples of container data types are: balanced and splay trees[1, 9, 10], skip lists[8], hash tables, stacks, queues and lists[2]. Each data type has unique operational and performance properties. For example, inserting an object into a stack is restricted to the stack top and takes constant time while inserting an object into a set of ordered objects represented by a balanced tree takes logarithmic time.

Container data structures are pervasive in programs and there are many library packages to deal with them. Most Unix/C environments include the functions *tsearch*, *hsearch*, *lsearch*, and *bsearch* to manipulate respectively objects stored in binary trees, hash tables, arrays and sorted arrays. C++ provides classes such as *Map* [3] and *Set* [5] to deal with ordered maps and unordered sets. Recently, a new set of C++ templates for ordered and unordered maps and sets called the Standard Template Library[6] has become increasingly popular.

Two common problems with existing container data type packages are interface confusion and performance deficiency. At the interface level, other than similar names, the Unix/C search functions have little else in common in how they manage objects. This is disconcerting because all container data types support the same basic set of abstract operations: *insert*, *delete*, *search*, and *iterate*. When such operations are realized via distinct interfaces and conflicting conventions, programmers have a hard time learning and programming them. The STL templates alleviate this problem by establishing interface guidelines for similar operations in different container data types. However, it is difficult with the STL templates to manage objects in multiple contexts because container and object types must be statically bound together to form dictionaries. Aside from interface issues, we shall see later that not only the older Unix/C packages but also the more modern C++ packages do not always perform well in both time and space usage.

This paper introduces *Cdt*, a C library for managing dictionaries based on common container data types: *list*, *stack*, *queue*, *ordered set*, *ordered multiset*, *unordered set* and *unordered multiset*. *Cdt* is unique among container packages in possessing the following characteristics:

- All dictionaries are manipulated via a uniform set of operations regardless of storage methods (i.e., container data types);
- Storage methods can be dynamically changed, for example, to turn an unordered dictionary to an ordered one;
- Object attributes are described in discipline structures that support both *set-like dictionaries*, i.e., dictionaries that identify objects by matching, and *map-like dictionaries*, i.e., dictionaries that identify objects by keys;
- Discipline structures can be dynamically

changed, for example, to change object comparators;

- Objects can be in multiple dictionaries, including dictionaries in shared memory; and
- Iterations are done directly over objects, i.e., no separate iterator types [6] needed.

*Cdt* uses splay trees for ordered sets and multisets, hash tables with move-to-front collision chains for unordered sets and multisets, and doubly linked lists for stacks, queues, and lists. The overall interface follows a *method* and *discipline* architecture [4, 11, 12]. Briefly, a library in this architecture provides handles and operations to hold and manipulate resource, methods to parameterize the semantics and performance characteristics of operations, and a discipline structure type that applications can use to define resource attributes and acquisition. Applying this architecture to *Cdt*, a handle is a dictionary and operations include handle creation, object insert, search, delete, etc. A *Cdt* method maps to a container data type while a discipline lets applications define information and operations that pertain directly to objects such as key type and object allocation.

## 2 The *Cdt* library

Objects are managed via three data types: dictionary, discipline, and method.

- *Dictionary*: A dictionary stores objects.
- *Method*: A dictionary has a method to define how objects are stored within it. Available methods are: *Dtset* for unordered sets, *Dtbag* for unordered multisets, *Dtoset* for ordered sets, *Dtobag* for ordered multisets, *Dtlist* for doubly linked lists, *Dtstack* for stacks and *Dtqueue* for queues.
- *Discipline*: Each dictionary has an application-defined discipline structure that specifies object comparison, hashing, allocation, and event announcement.

### 2.1 Dictionary operations

This section briefly overviews the main functions in the *Cdt* library.

*Dt\_t\* dtopen(Dtdisc\_t\* disc, Dtmethod\_t\* meth)* creates a dictionary of type *Dt\_t* with the given discipline *disc* and method *meth*. A dictionary *dt* is closed or cleared with *dtclose(Dt\_t\* dt)* and *dtclear(Dt\_t\* dt)*.

Functions *Void\_t\* dtsearch(Dt\_t\* dt, Void\_t\* obj)* and *Void\_t\* dtmatch(Dt\_t\* dt, char\* key)* search the dictionary *dt* for an object matching respectively *obj* or *key*. Such a matched object becomes a *current object* with special semantics in certain operations discussed below. *Void\_t* is defined as *void* for ANSI-C or C++ and *char* for older C variants so it is suitable for exchanging addresses between the library and applications.

*Void\_t\* dtinsert(Dt\_t\* dt, Void\_t\* obj)* inserts an object *obj* into the dictionary *dt*. Methods *Dtset* and *Dtoset* allow *obj* to be inserted only if there is no matching object already in *dt*. Other methods always insert a new object because they allow insertion of equal objects. Method *Dtstack* inserts objects at stack top. Method *Dtqueue* inserts objects at queue tail. Method *Dtlist* inserts an object before the current object of *dt* if there is one, or at list head otherwise. An inserted or found object becomes the new current object.

*Void\_t\* dtdelete(Dt\_t\* dt, Void\_t\* obj)* is used to delete from *dt* an object matching *obj* if one exists. *dtdelete(dt, NULL)* works with *Dtstack* and *Dtqueue* and removes respectively the top or head object.

Object iteration depends on a particular object ordering defined by the storage method in use. For *Dtoset* and *Dtobag*, objects are ordered by object comparisons. For *Dtstack*, objects are ordered in the reverse order of insertion. For *Dtqueue*, objects are ordered in the order of insertion. For *Dtlist*, objects are ordered by their list positions. For *Dtset* and *Dtbag*, the object order is defined at the point of use and may change on any search or insert operation.

There are many ways to iterate over objects in a dictionary. The below loop iterates forward over all objects in a dictionary *dt*:

```
for(o = dtfirst(dt); o; o = dtnext(dt,o) )
```

Alternatively, the below loop can be used to iterate backward over objects:

```
for(o = dtlast(dt); o; o = dtprev(dt,o) )
```

## 2.2 Storage methods

A storage method is of type `Dtmethod_t` and defines how objects are manipulated. *Cdt* provides the following storage methods:

- **Dtset and Dtbag:** These methods are based on hash tables with move-to-front collision chains. *Dtset* stores unique objects while *Dtbag* allows repeatable objects (i.e., objects that compare equal). Repeatable objects are collected together so that any iteration always passes over sections of them. Object accesses take expected  $O(1)$  time given a good hash function.
- **Dtoset and Dtobag:** These methods store ordered objects in top-down splay trees. *Dtoset* stores unique objects while *Dtobag* allows repeatable objects. Object accesses take amortized  $O(\log n)$  time. Splay trees adapt well to biased access patterns because frequently accessed objects migrate closer to tree roots.
- **Dtlist:** This method stores repeatable objects in a doubly-linked list. An object is always inserted in front of the current object which is either the list head or established by a search, insert, or iteration. Object insertion and deletion are done in  $O(1)$  time.
- **Dtstack and Dqueue:** These methods store repeatable objects in stack and queue order. In a stack order, objects are kept in reverse order of their insertion. In a queue order, objects are kept in order of their insertion. Object insertion and deletion are done in  $O(1)$  time.

## 2.3 Disciplines

A discipline structure is of type `Dtdisc_t`. Applications use disciplines to define object attributes such as comparison, hashing, and allocation.

Figure 1 shows `Dtdisc_t`. `Dtdisc_t.key` and `Dtdisc_t.size` identify a key of type `Void_t*` used for object comparison or hashing. `Dtdisc_t.key` defines the offset in an object where the key resides. `Dtdisc_t.size` defines the key type. A positive value means that the key is a byte array of given length, a zero value means that the key is a null-terminated string, and a negative value means that the key is a null-terminated string whose address is stored at the key offset.

```
typedef struct
{ int      key;      /* key offset */
  int      size;     /* key size/type */
  int      link;     /* object holder */
  Dtmake_f makef;    /* object makef */
  Dtfree_f freef;    /* object freef */
  Dtcompar_f comparf; /* comparator */
  Dthash_f hashf;    /* hash function */
  Dtmemory_f memoryf; /* allocator */
  Dtevent_f eventf;  /* event handler */
} Dtdisc_t;
```

Figure 1: A discipline structure

Objects are held in a dictionary via holders of type `Dtlink_t`. If `Dtdisc_t.link` is negative, the library will allocate object holders. Otherwise, the library assumes that object holders are embedded inside objects and `Dtdisc_t.link` defines the offset in an object where the holder resides.

`Dtdisc_t.makef` and `Dtdisc_t.freef`, if defined, are called to make and free objects when they are inserted or deleted. If `Dtdisc_t.makef` is not defined, then in the call `dtinsert(dt,obj)` `obj` itself will be inserted.

If `Dtdisc_t.comparf` or `Dtdisc_t.hashf` are not defined, some internal functions are used. By allowing both key definition and compare function in a discipline, both set-like and map-like dictionaries are supported.

`Dtdisc_t.memoryf`, if defined, is used to allocate space. `Dtdisc_t.eventf`, if defined, announces various events such as dictionary opening and closing and method or discipline changes.

## 2.4 An example *Cdt* application

A common container data type example is given in the *Map* associative array paper [3] and the Unix/C manual page for the function `tsearch()`. This application reads a text file, partitions it into tokens (strings separated by space, tab and new line characters), keeps frequency count for each token, and finally writes out the tokens and their frequencies.

Figure 2 shows an implementation of the token counting example. Omitted are a few minor grammatical statements and the function `readtoken()` to parse an input stream into tokens. The below comments are based on line numbers in the figure:

```

1. #include    <sfio.h>
2. #include    <cdt.h>

3. typedef struct
4. {   Dtlink_t link;
5.     char*   token;
6.     int      freq;
7. } Token_t;

8. Dtdisc_t Tkdisc =
9. {   offsetof(Token_t,token), -1, 0 };

10. Token_t* newtoken(char* s)
11. {   Token_t* tk;
12.     tk = malloc(sizeof(Token_t));
13.     tk->token = malloc(strlen(s)+1);
14.     strcpy(tk->token,s);
15.     tk->freq = 1;
16.     return tk;
17. }

18. main()
19. {   char*   s;
20.     Token_t* tk;
21.     Dt_t*   dt = dtopen(&Tkdisc,Dtset);

22.     while((s = readtoken(sfstdin)) )
23.     {   if((tk = dtmatch(dt,s)) )
24.         tk->freq += 1;
25.         else dtinsert(dt,newtoken(s));
26.     }

27.     for(tk = dtfirst(dt); tk;
28.         tk = dtnext(dt,tk) )
29.         fprintf(sfstdout,"%s:\t%d\n",
30.             tk->str, tk->freq);
31. }

```

**Figure 2: Program to count tokens**

- 1-2: The header file `sfio.h` [4] declares I/O functions. `cdt.h` is the *Cdt* public header file and declares necessary types, values and functions.
- 3-7: `Token_t` is a structure to hold a string token and a frequency count `freq`. It also embeds the container holder structure in the `link` field.
- 8-9: The discipline `Tkdisc` describes attributes of `Token_t` objects. The ANSI-C macro `offsetof()` defines the offset of `Token_t.token` in `Token_t`. Since `Token_t.token` points to a null-terminated string, `Tkdisc.size` is set to `-1`. `Tkdisc.link` is set to 0, the offset to `Token_t.link` in `Token_t`.
- 10-17: `newtoken()` is a function to create a new `Token_t`

structure from a given string `s`. To simplify the exposition, error checks for the `malloc` calls were omitted.

- 21: A new dictionary `dt` is created based on the discipline `Tkdisc` and the method `Dtset`. Here it is assumed that tokens need not be sorted. Otherwise, `Dtoset` could be used (see also Section 4).
- 22-26: Tokens are read and inserted into `dt`. Line 22 uses `dtmatch()` to find out if a token matching the current read token already exists in `dt`. In that case, only its frequency count is updated. Otherwise, Line 24 creates and inserts a new token structure into `dt`.
- 27-30: These lines loop over all tokens and output both tokens and their frequency counts. Note that this is done directly over objects without the aid of any iterator type [6].

## 3 Performance

Among the various container data types, hash tables and binary trees are most common and also have large variation in implementation quality. This section presents results from a performance study that compared various set and map container structure packages based on hash tables and binary trees.

### 3.1 Methodology

The token counting application in Section 2.4 was used as a benchmark. To minimize implementation variation, a single program based on Figure 2 was written. Compile time options allowed switching usages of the *Cdt* methods `Dtoset` and `Dtset`, the Unix/C package `tsearch`, the C++ classes `Set` and `Map`, and the STL templates `map` and `hashmap`. All implementations used the same string comparison function, a variant of `strcmp()` that also keeps invocation count. In addition, all hash table implementations used the same hash function supplied by *Cdt* so that hash value computation would be uniform. This was necessary because the default hash functions in some of the packages were not very good. For example, the comparison counts in Section 3.2 for the `Set` package would have been much higher if its default hash function was used.

A variety of input files were used:

- *ps*: PostScript source of a technical paper,
- *src*: an archive of C source code,
- *kjv*: a King James version of the bible,
- *mbox*: a personal mail archive,
- *host*: a database mapping IP addresses to machine hosts, and
- *city*: a database mapping cities to area codes.

| File        | Size   | Tokens  | Distinct | Length |
|-------------|--------|---------|----------|--------|
| <i>ps</i>   | 1,989K | 335,997 | 11,912   | 38.00  |
| <i>src</i>  | 1,169K | 149,886 | 27,964   | 16.40  |
| <i>kjv</i>  | 4,441K | 822,587 | 33,916   | 8.01   |
| <i>mbox</i> | 2,701K | 419,197 | 49,903   | 9.83   |
| <i>city</i> | 1,349K | 81,206  | 69,610   | 18.17  |
| <i>host</i> | 2,722K | 449,554 | 102,566  | 16.71  |

**Table 1: Summary of benchmark input files**

Table 1 summarizes input file statistics: file size in K-bytes, total number of tokens, number of distinct tokens, and average length of a token. These input files represent a wide variety of data ranging from *ps* which has relatively few distinct tokens to *city* which has about 85% distinct tokens. Tokens in *host* and *city* are also highly ordered.

| Program            | Size    |
|--------------------|---------|
| <b>Set</b>         | 146,632 |
| <b>hashmap</b>     | 179,988 |
| <b>map</b>         | 145,892 |
| <b>Map</b>         | 83,795  |
| <b>tsearch</b>     | 66,744  |
| <b>Dtset+Dtset</b> | 73,712  |

**Table 2: Sizes of benchmark programs**

The experiment was performed on a SPARC-20 running SUN OS5.0. Table 2 shows the sizes of the benchmark programs. Both *Dtset* and *Dtset* were combined in the same benchmark program with invocation options for method selection. Except for *Map*, other C++ packages caused the test code to be about twice as large as the C versions, a sign of code bloating due to the use of templates. Comparing the results of compiling `main(){} with C and C++ showed that only about 5K can be attributed to language difference.`

Program execution was done at night on a quiescent machine. Each time measurement was obtained by

running the same test 9 times, computing total cpu and system times for each run, discarding the top two and bottom two scores to reduce variance, then averaging the remaining five scores. Space measurements were done by calling `sbrk(0)` before any dictionary was opened and after all output was done and computing differences in the return values.

### 3.2 Hash table packages

Below are brief descriptions of the container packages that use hash tables to implement unordered sets and maps. The Unix/C *hsearch* package was omitted because it was too slow to measure.

- **Set**: The C++ *Set* class that comes standard with our compiler. This uses a hash table with chaining to resolve collisions.
- **hashmap**: The C++ STL *hashmap* template. This uses a hash table with chaining to resolve collisions.
- **Dtset** Method *Dtset* of *Cdt*. This uses a hash table with chaining. The collision chains use a move-to-front heuristic to improve search time.

| Dataset     | Set | hashmap | Dtset |
|-------------|-----|---------|-------|
| <i>ps</i>   | 328 | 682     | 324   |
| <i>src</i>  | 131 | 566     | 122   |
| <i>kjv</i>  | 799 | 4,038   | 789   |
| <i>mbox</i> | 386 | 1,839   | 370   |
| <i>city</i> | 35  | 292     | 12    |
| <i>host</i> | 384 | 2,279   | 347   |

**Table 3: Hash: comparison counts in thousands**

Table 3 shows comparison counts for the hash table packages in units of thousands. *hashmap* performed worst, with comparison counts many times higher than that of *Set* and *Dtset*. *Dtset* asserted that tokens compared equal must have the same hash values. This fact was used effectively to reduce many comparisons because the hash function distinguished objects well. The low comparison counts for *Set* suggested that it might have used the same strategy as *Dtset*. *Dtset* retains a slight edge perhaps due to its move-to-front strategy on collision chains.

Table 4 shows time performance. Poor comparison counts directly translated to poor computing time. *hashmap* was worst, sometimes up to a factor of 3

| Dataset     | Set   | hashmap | Dtset |
|-------------|-------|---------|-------|
| <i>ps</i>   | 7.47  | 10.76   | 4.32  |
| <i>src</i>  | 6.06  | 7.57    | 3.59  |
| <i>kju</i>  | 20.61 | 31.49   | 11.01 |
| <i>mbox</i> | 14.27 | 19.84   | 8.20  |
| <i>city</i> | 9.49  | 10.66   | 6.49  |
| <i>host</i> | 18.28 | 23.31   | 10.78 |

Table 4: Hash: times in seconds

| Dataset     | Set    | hashmap | Dtset |
|-------------|--------|---------|-------|
| <i>ps</i>   | 1,464  | 976     | 912   |
| <i>src</i>  | 2,752  | 1,432   | 1,496 |
| <i>kju</i>  | 3,048  | 1,448   | 1,584 |
| <i>mbox</i> | 4,632  | 2,232   | 2,360 |
| <i>city</i> | 7,144  | 3,688   | 3,944 |
| <i>host</i> | 10,552 | 5,248   | 5,496 |

Table 5: Hash: space in K-bytes

slower than Dtset. Dtset was fastest among the three packages.

Table 5 shows space usage. Dtset and hashmap used about the same amount of space. Set sometimes used twice as much space as the other packages.

### 3.3 Binary tree packages

A further requirement could be stimulated in the token counting example that tokens must be output in a lexicographic order. In that case, a natural solution is to use container packages that maintain ordered tokens. Below are the studied container packages for ordered sets and maps:

- **tsearch:** The **tsearch** function in SUN OS5.4. This uses plain binary trees.
- **Map:** The C++ Map class. This is based on AVL balanced trees.
- **map:** The C++ STL map template. This uses red-black balanced trees.
- **Dtset:** Method Dtset of *Cdt*. This uses top-down splay trees.

Table 6 shows comparison counts for the binary tree methods. Except for *city* and *host*, **tsearch** performed well despite its simplistic data structure. This is because most datasets consist of more or less random tokens and binary trees built from such random data are naturally balanced. **tsearch** did poorly

| Dataset     | Map    | map    | tsearch | Dtset |
|-------------|--------|--------|---------|-------|
| <i>ps</i>   | 14,267 | 10,434 | 7,314   | 1,636 |
| <i>src</i>  | 5,857  | 4,553  | 3,533   | 1,773 |
| <i>kju</i>  | 28,142 | 26,119 | 13,526  | 8,592 |
| <i>mbox</i> | 15,747 | 13,229 | 7,396   | 5,425 |
| <i>city</i> | 3,841  | 3,006  | 12,189  | 1,631 |
| <i>host</i> | 22,963 | 15,657 | 19,750  | 2,568 |

Table 6: Tree: comparison counts in thousands

| Dataset     | Map   | map   | tsearch | Dtset |
|-------------|-------|-------|---------|-------|
| <i>ps</i>   | 15.37 | 15.51 | 13.26   | 5.33  |
| <i>src</i>  | 9.63  | 9.05  | 8.20    | 5.53  |
| <i>kju</i>  | 37.08 | 44.79 | 26.77   | 21.27 |
| <i>mbox</i> | 24.22 | 28.13 | 17.86   | 15.85 |
| <i>city</i> | 12.01 | 11.58 | 25.60   | 8.92  |
| <i>host</i> | 30.19 | 26.58 | 61.00   | 13.76 |

Table 7: Tree: times in seconds

on *city* and *host* whose tokens were highly ordered. The balanced tree packages **Map** and **map** ignored any such ordering property in the data. Both packages used about the same number of comparisons with **map** having a slight edge. The splay tree approach in **Dtset** took advantage of data ordering to reduce comparisons. As a result, **Dtset** was the clear winner in all cases.

Table 7 shows time performance. As with the hash table methods, comparison counts mapped directly to time. **Dtset** was fastest, sometimes by a factor of 3 or more over some of the other methods. Note that **Dtset** was even faster than the STL **hashmap** package which did not have to order tokens.

Table 8 shows memory usage. **tsearch** and **Map** used more memory than other methods. **Map**'s extra space was due to the balancing data. The cause for **tsearch**'s poor memory usage was unclear although a memory trace using *Vmalloc* [12] revealed a mysterious extra allocation after each holder allocation.

| Dataset     | Map   | map   | tsearch | Dtset |
|-------------|-------|-------|---------|-------|
| <i>ps</i>   | 1,064 | 872   | 1,064   | 880   |
| <i>src</i>  | 1,864 | 1,424 | 1,872   | 1,432 |
| <i>kju</i>  | 1,968 | 1,440 | 1,976   | 1,440 |
| <i>mbox</i> | 3,008 | 2,224 | 3,016   | 2,232 |
| <i>city</i> | 4,768 | 3,672 | 4,776   | 3,680 |
| <i>host</i> | 6,840 | 5,240 | 6,856   | 5,240 |

Table 8: Tree: space in K-bytes

## 4 Flexible programming with *Cdt*

To output tokens in order, a strategy that often works better than just using *Dtset* is as follows. First, *Dtset* is used to construct the token dictionary. Then, *Dtset* is used right before outputting to sort tokens into the right order. To implement this strategy, the below line of code can be inserted before Line 26 of Figure 2:

```
dtmethod(dt,Dtset);
```

| Dataset     | Dtset | Dtset | Dtset+Dtset |
|-------------|-------|-------|-------------|
| <i>ps</i>   | 324   | 1,636 | 498         |
| <i>src</i>  | 122   | 1,773 | 608         |
| <i>kju</i>  | 789   | 8,592 | 1,307       |
| <i>mbox</i> | 370   | 5,425 | 1,282       |
| <i>city</i> | 12    | 1,631 | 1,420       |
| <i>host</i> | 347   | 2,568 | 2,014       |

**Table 9: Tuning: comparison counts in thousands**

Table 9 shows comparison counts for the above strategy. Except for *city* and *hosts*, *Dtset+Dtset* improves substantially over the exclusive use of *Dtset*, up to 70% for *kju*.

| Dataset     | Dtset | Dtset | Dtset+Dtset |
|-------------|-------|-------|-------------|
| <i>ps</i>   | 4.32  | 5.33  | 4.83        |
| <i>src</i>  | 3.59  | 5.53  | 5.12        |
| <i>kju</i>  | 11.01 | 21.27 | 12.50       |
| <i>mbox</i> | 8.20  | 15.85 | 11.55       |
| <i>city</i> | 6.49  | 8.92  | 12.73       |
| <i>host</i> | 10.78 | 13.76 | 18.38       |

**Table 10: Tuning: times in seconds**

Table 10 show time performance. Time usages for *Dtset+Dtset* markedly improved over the lone use of *Dtset* on most datasets except for *city* and *hosts*. For these datasets, though comparison counts went down somewhat, time measurements actually went up. This was because these datasets contained many distinct tokens and *Dtset* ended up repeating *Dtset*'s work.

The above situation is common in practice. Programs must often deal with data that have special characteristics. It is seldom the case that efficient algorithms can be devised to adapt smoothly to the data diversity and operate optimally in each special situation. Therefore, whenever possible, a good design principle is to let users select and combine

```
1. int freqcmp(Dt_t* dt, Void_t* arg1,
2.             Void_t* arg2, Dtdisc_t* disc)
3. {   int    d;
4.     Token_t* t1 = (Token_t*)arg1;
5.     Token_t* t2 = (Token_t*)arg2;
6.     if((d = t1->freq - t2->freq) != 0)
7.         return d;
8.     else return strcmp(t1->token,t2->token);
9. }

10. Tkdisc.comparf = freqcmp;
11. Tkdisc.key = Tkdisc.size = 0;
12. dtdisc(dt,&Tkdisc,DT_SAMEHASH|DT_SAMECMP);
13. dtmethod(dt,Dtset);
```

**Figure 3: Order tokens by frequency**

computing methods to optimize processing based on specific knowledge of the data. *Cdt* simplifies doing this in the context of using container data types. In fact, the benchmark program was written to allow strategy selection at invocation time. It is not easy to do the same using the other container packages.

As another example of *Cdt*'s flexibility, suppose that the output requirement is changed to ordering tokens in increasing order of frequency. To do this, Figure 2 should be augmented with Lines 1-9 of Figure 3 before *main()* and Lines 10-13 of the same figure before Line 26. The below comments pertain to line numbers in Figure 3:

- 1-9: The function *freqcmp()* compares tokens first by frequency, then by token names. So within a group of tokens with the same frequency, tokens will be ordered lexicographically.
- 10: The comparator is redefined to be *freqcmp()*.
- 11: *Tkdisc.key* and *Tkdisc.size* are set to 0 to indicate that *Token\_t* objects will be compared whole instead of via the key strings *Token\_t.token*.
- 12: *dtdisc()* is called to officially change the discipline. Normally, a discipline change implies rearranging of objects because hash values may have changed or objects that used to compare distinct may have become equal. The flags *DT\_SAMEHASH* and *DT\_SAMECMP* tell *dtdisc()* that, in this case, both hash values and object comparison remain unchanged. The latter is strictly untrue but it saves computation that would be done anyway on line 13.

13: `dtmethod(dt,Dtset)` is called to switch the storage method to `Dtset` and sort tokens by the new comparator.

Note that in this example it is possible to use method `Dtset` with the comparator `freqcmp()` from the start of the application. However, doing so would have been prohibitively expensive because objects must be deleted and reinserted each time their frequencies are updated. Thus, for efficiency, it is necessary that `Dtset` is used during dictionary construction and `Dtset` is used only at the end before outputting.

## 5 Discussion

This paper introduced *Cdt*, a container data type library. The library provides the common storage methods: *set*, *multiset*, *ordered set*, *ordered multiset*, *list*, *stack* and *queue* which are often seen only in isolated packages. *Cdt* achieves an interface that keeps orthogonal the three design dimensions: dictionary operations, storage methods, and object descriptions. This is a goal attempted but not quite achieved by other recent work on reusable components such as the C++ Standard Template Library.

Many contemporary container libraries are unwieldy because their interfaces are not sufficiently abstract and operations are tied too closely to container data types. At worst, this leads to divergent interfaces for the same basic operations as shown by the Unix/C search functions. Even with better interface design as in the STL case, the close tie between implementation techniques and abstract interfaces can reduce the generality of the library. For example, instead of a general container template that can be parameterized by storage methods, STL provides various container templates such as `hashmap` and `map` that are strongly bound to minimal object requirements according to the respective implementation techniques. As a result, although `hashmap` and `map` provide similar functions they require objects with different type specifications. This means that there is no simple way to dynamically convert a `hashmap` container to a `map` container in the style discussed in Section 4. *Cdt* avoids such interface limitations by making dictionary operations completely abstract and parametrizable by methods and disciplines which are orthogonal and mutable attributes of dictionaries. The method and discipline architecture naturally lifts a library interface to its most general level. Perhaps

some future STL work can benefit from such an interface analysis and design.

*Cdt* disciplines are run-time structures used to define object attributes such as keys, comparison, hashing, and allocation. By allowing both comparators and keys, *Cdt* generalizes set-like and map-like container packages. This leads to a unifying interface to manage such containers. Using run-time structures for type definition means losing certain services common to C++ templates such as static type checking and inlining of comparison functions. The loss of static type checking is balanced out by the added programming flexibility. For example, *Cdt* allows the same objects to be described in multiple ways and both disciplines (i.e., object types) and methods (i.e., container types) can be arbitrarily mixed and changed. The efficiency loss resulted from no inlining of object comparisons is compensated by the advantage of having a single library code image and consequent code size reduction as exemplified in Table 2. A single code image also makes possible using *Cdt* as a dynamically loadable shared library. Further, for applications such as the discussed token counting example which require relatively complex objects, any function call overhead to compare two objects would be negligible relative to the cost of the comparison itself.

A performance study showed that *Cdt* methods `Dtset` and `Dtset` performed as well or better than their counterparts in other C and C++ container libraries including the modern STL components. The *Cdt* methods consistently used about the same or less space than other packages while they were faster than other packages by up to a factor of two or more. The use of splay trees and hash tables with self-adjusting collision chains enable these methods to perform well in a wide range of input data. Examples were given showing how further performance gains can be made with selective matching of disciplines and methods at run time.

*Cdt* is a descendant of *Libdict* [7]. It is a mature library and has been used in many applications including large-scale information systems that routinely handle dictionaries with tens to hundreds thousands of objects.

## Acknowledgement

*Cdt* evolved over many years and benefited from advice and demands of many friends and users. In particular, I'd like to thank Glenn Fowler, David

Korn, and Stephen North who patiently survived several generations of interface changes. In addition, I'd like to thank Carl Staelin whose careful reading of this paper helped improving it greatly.

## Code availability

*Cdt* source code is available at:

<http://www.research.att.com/sw/tools/reuse/>.

## REFERENCES

- [1] G.M. Adelson-Velskii and E.M. Landis. An Algorithm for the Organization of Information. *Soviet Math. Doklady*, 3:1259–1263, 1962.
- [2] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 1973.
- [3] Andrew R. Koenig. Associative Arrays in C++. In *Proceedings of Summer 1988 USENIX Conference*, pages 173–186, 1988.
- [4] David G. Korn and Kiem-Phong Vo. SFIO: Safe/Fast String/File IO. In *Proc. of the Summer '91 Usenix Conference*, pages 235–256. USENIX, 1991.
- [5] Unix System Laboratories. *USL C++ Standard Components Programmer's Reference*. AT&T and Unix System Laboratories, Inc., 1990.
- [6] David R. Musser and Atul Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, 1995.
- [7] Stephen C. North and Kiem-Phong Vo. Dictionary and Graph Libraries. In *Proc. of the Winter '93 Usenix Conference*, pages 1–11. USENIX, 1993.
- [8] T. Papadakis. *Skip Lists and probabilistic Analysis of Algorithms*. University of Waterloo, 1993.
- [9] Robert Sedgewick. *Algorithms, 2nd Edition*. Addison-Wesley, 1988.
- [10] D. Sleator and R.E. Tarjan. Self-Adjusting Binary Search Trees. *JACM*, 32:652–686, 1985.
- [11] Kiem-Phong Vo. Writing Reusable Libraries with Disciplines and Methods. In *Practical Reusable UNIX Software*. John Wiley & Sons, 1994.
- [12] Kiem-Phong Vo. Vmalloc: A General and Efficient Memory Allocator. *Software Practice & Experience*, 26:1–18, 1996.



# A Simple and Extensible Graphical Debugger

David R. Hanson and Jeffrey L. Korn

*Department of Computer Science, Princeton University,  
35 Olden St., Princeton, NJ 08544  
{drh,jlk}@cs.princeton.edu*

## Abstract

*deet* is a simple but powerful debugger for ANSI C and Java. It differs from conventional debuggers in that it is machine-independent, graphical, programmable, distributed, extensible, and small. Low-level operations are performed by communicating with a “nub,” which is a small set of machine-dependent functions that are embedded in the target program at compile-time, or are implemented on top of existing debuggers. *deet* has a set of commands that communicate with the target’s nub. The target and *deet* communicate by passing messages through a pipe or socket, so they can be on a different machines. *deet* is implemented in *tksh*, an extension of the Korn shell that provides the graphical facilities of Tcl/Tk. Users can browse source files, set breakpoints, watch variables, and examine data structures by pointing and clicking. Additional facilities, like conditional breakpoints, can be written in either Tcl or the shell. Most debuggers are large and complicated, *deet* is less than 1,500 lines of shell plus a few hundred lines of machine-specific nub code. It is thus easy to understand, modify, and extend. We describe an implementation of the nub API for Java and an implementation that is layered on top of *gdb*. We have also implemented a version of *gdb* using the nub API, which demonstrates the modularity of the design.

## 1 Introduction

Traditional UNIX debuggers are indispensable tools for locating and fixing program errors. Despite their importance and pervasiveness, they continue to harbor inadequacies that limit their usability. For example, UNIX debuggers typically have textual user interfaces that are cryptic at best. When debuggers are hard to use, programmers tend to litter their programs with print statements instead of using a debugger.

While most PC debuggers run on only one operating system and one architecture, UNIX debuggers must deal with portability issues. Debuggers are notoriously machine-dependent programs; they depend on the tar-

get architecture, operating system, compiler, and linker. Thus, porting a debugger from one variant of UNIX to another can require a substantial amount of effort. For example, about one-third of *gdb*’s source code is machine-dependent.

Few debuggers have programming facilities in which, for example, programmers can write application-specific debugging code. Such code is useful for nontrivial queries of data structures, such as displaying the second to last element in a linked list, or all the positive elements in an array. Other examples include setting conditional breakpoints and automating program testing. Debuggers that support programming facilities do exist, but often the language is idiosyncratic to either the debugger or the source language, or both, and hard to learn.

Most debuggers are large and complex programs; for example, *gdb* [14] is about 150,000 lines of C. This complexity has some unfortunate consequences. First, debuggers are often themselves buggy, because, like any large program, their complexity and size makes them prone to errors and to inconsistent behaviors on different platforms. Second, debuggers are usually difficult to extend, because their implementations may be hard to understand and to modify.

*deet* (*desktop error elimination tool*) addresses these shortcomings. It provides both textual and graphical interfaces to make it easy to use. Users can perform most debugging actions by pointing and clicking, and data structures can be displayed graphically. The GUI is written with Tk [12]. *deet* is also programmable: Its capabilities can be extended by writing in either Tcl or in *tksh*, a variant of the Korn shell [8].

Nearly all of *deet*’s implementation is machine-independent. It uses a small “nub” that provides facilities for communicating with the debugger and controlling the target. The nub-based approach permits *deet* to debug a target running on another machine. Figure 1 shows the screen of a typical debugging session. *deet* doesn’t attempt to match debuggers like *gdb* feature-for-feature; for example, *deet* can’t examine core dumps, evaluate arbitrary C expressions, or debug at the assembly-language level. Nevertheless, its implementation is sur-

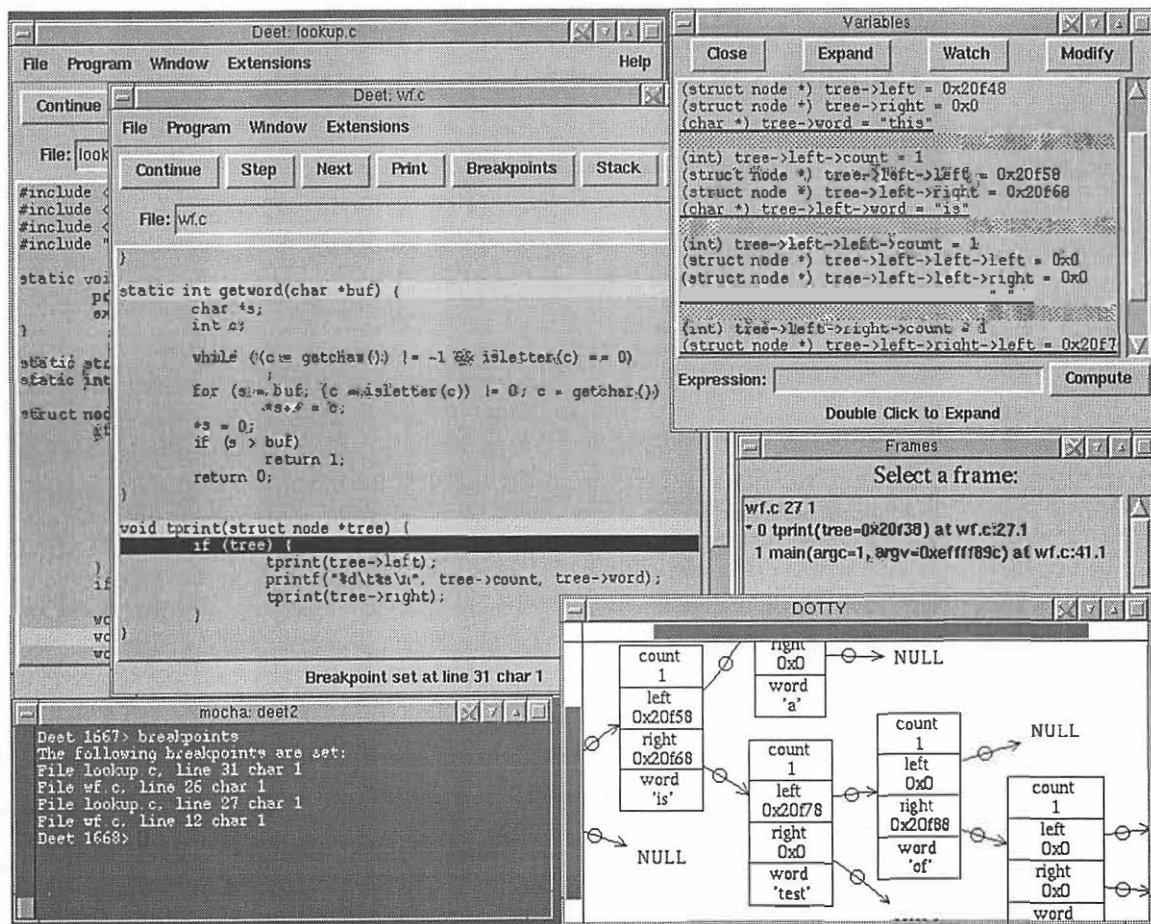


Figure 1: deet screen dump

prisingly simple. Its complete source is approximately 2,500 lines of shell and C.

## 2 Using deet

deet's features are best explained by seeing it in action. First, the target program is compiled by `lcc` [3] with the appropriate debugging option to embed the nub in the target:

```
$ lcc -Wf-g4 wf.c lookup.c
```

Here and in the displays below, *slanted* type identifies user input. When the generated `a.out` is executed, the debugger specified by the environment variable `DEBUGGER` is also started, so

```
$ DEBUGGER=deet a.out
```

starts both `a.out` and `deet`. At this point, the source window shown in Figure 2 appears. The user is prompted for textual `deet` commands in the shell window from

which a `.out` was invoked, but most debugging actions are performed with the mouse.

Single-clicking on a line highlights that line if it contains a breakpoint; double-clicking on the line sets the breakpoint. `deet` can set breakpoints on expressions, not just statements, so there may be more than one breakpoint in a line. When a line has multiple breakpoints, double-clicking sets the breakpoint closest to the cursor. Breakpoints are indicated in the window in lighter shading or in yellow (see Figure 2). Double-clicking on a breakpoint that has already been set removes the breakpoint.

The breakpoints window, like the one shown in Figure 3, displays a list of all breakpoints and related information about each breakpoint, such as its location and break condition. These conditions are `deet` expressions that are evaluated whenever the breakpoint is reached; if the condition is true, the target stops. When the target stops at a breakpoint, the current source window shows the file and line number of the breakpoint, and reverse video highlights the line containing the breakpoint.

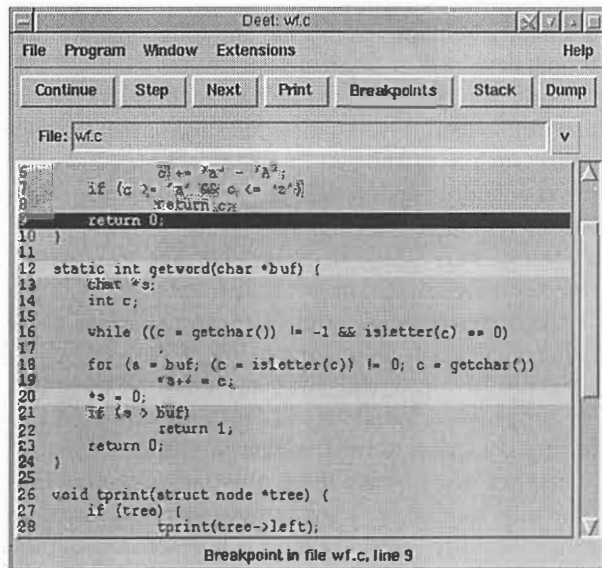


Figure 2: Source window

A condition can be changed by highlighting the breakpoint in the breakpoints window and editing the condition field, and a breakpoint can also be removed by clicking the "Delete" button in this window.

The stack can be shown by clicking on the "Stack" button in the source window (see Figure 2). This displays a new window that shows each frame on the stack, from the top down, as illustrated in the right middle portion of Figure 1. An individual frame can be selected, and clicking a button in the stack window performs the corresponding action on that frame. For instance, by clicking the "Dump" button, the names and values of the parameters and locals for that frame are displayed. Clicking the "OK" button causes the source window to display the file and line number of the call to the selected frame.

Highlighting a variable in the source window and clicking "Print" causes a pop-up window to display the value of that variable (see the upper right corner of Figure 2). If the variable is a pointer, a structure, or a union, double-clicking on the variable expands its value. For pointers, the value of the referent is displayed; for structures and unions, the values of the fields are displayed. *deet* also displays the values of variables in balloon help pop-up windows when the cursor is left on top of the variables for sufficient time, similar to Microsoft's Visual C++ debugger [11].

A variable can be modified by clicking "Modify" in the variable window, which prompts the user to enter a new value. A variable may also be watched, which causes its value to be displayed in the variable window and updated as execution passes each potential breakpoint.

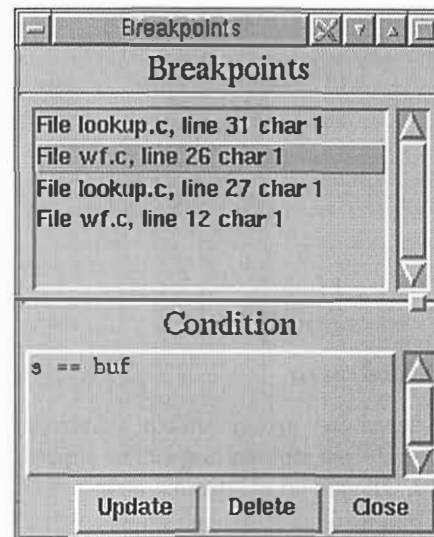


Figure 3: Breakpoints window

Commands may be typed at the debugger in the shell window in a manner similar to *gdb*. For instance, the *breakpoints* command displays the current breakpoints:

```
deet> breakpoints
The following breakpoints are set:
File test/wf.c, line 4 char 28
File test/lookup.c, line 14 char 50
```

Commands are just *tksh* commands, so shell commands like *history*, *pwd*, and *make* can be entered as well.

Most of the state in a *deet* debugging session can be saved and restored later in a subsequent, separate debugging session. This state includes breakpoints and their conditions, locations of files, and user-defined *tksh* functions. *deet* saves the state by writing a shell script that can be interpreted to restore the state.

### 3 Design

*deet* divides cleanly into two parts: One part interacts with the programmer, and the other part interacts with the target program. The user-interface part is written in *tksh*, a version of the new Korn shell [2, 7] that has been extended to support *Tcl* [12]. The target program is controlled by a *nub*, which provides debugging primitives, as detailed below. *deet*'s implementation of and interaction with the *nub* is also written in *tksh*. Thus, programmers can modify and extend *both* parts of *deet* by writing *tksh* code.

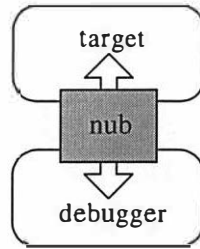


Figure 4: cdb's design

### 3.1 Cdb and deet

deet is based on cdb [6]. cdb is a machine independent debugger that eliminates machine dependencies by adding a small amount of information into the target program at compile time. cdb communicates with the target through a nub—a small machine-independent interface that constitutes the core functionality of the debugger, as suggested by Figure 4. The nub implementation can be made machine-independent, as cdb shows, but machine-dependent implementations are also possible and are undoubtedly more efficient. The nub is small enough that re-implementing it for new platforms is nearly as easy as porting the machine-independent implementation.

There are four components of cdb:

1. The nub interface, which stands between the debugger and machine-dependent target manipulations.
2. The nub implementation, which consists of the nub interface functions, special code emitted by the compiler to support the nub, and a wrapper around the linker to load the nub and the machine-independent symbol table.
3. A machine-independent symbol table format, which is emitted by the compiler and linked into the target.
4. A simple, text-based debugger that uses the nub to provide minimal functionality; this debugger is intended to be replaced with more sophisticated debuggers, like deet.

Any of the last three components can be replaced with alternative implementations. For instance, the nub can be replaced with a machine-dependent implementation that uses the `ptrace` system call like most UNIX-specific debuggers, or by one that is layered on top of `gdb`. The machine-independent symbol tables could be replaced with the usual machine-dependent “stab” symbol tables embedded in UNIX executables. Finally, the debugger itself could be replaced with any program that uses the nub interface. deet is a replacement for this fourth component. Using deet does not directly involve changes to

any of the other components, but implementing deet did induce additions to the nub and to the symbol-table format beyond their original designs.

deet is written in tksh, which includes a C library that can be used to manipulate the state of the Tcl interpreter, such as reading and writing variables and creating new built-in commands. tksh can run any library written on top of the Tcl library, which includes the Tk graphics library. Thus, Tk commands, like `button` and `pack`, can be invoked from tksh scripts.

tksh should be thought of as an extension to Tcl rather than as an alternative to it. tksh allows Tcl scripts to be run directly with the `source` command. Tcl scripts share variables and functions with tksh, allowing Tcl scripts to work with shell scripts.

tksh is used as the debugging language for deet primarily because of its strengths as an interactive command language. Debuggers are interactive programs. deet takes advantage of the interactive facilities of tksh, such as command-line editing, job control and pipelines. Using the command-line interface to deet feels like using a shell because the debugger itself is an extension of ksh. tksh also offers two familiar, high-level languages. Many programmers already know how to write shell and Tcl scripts, which is 90% of what's needed to use deet. However, a perl or python programmer could rewrite the deet front end and still use existing nub implementations.

deet also includes additional built-in commands for debugging. deet's code is simpler to understand than the corresponding C code would be, because it's written in a high-level language. deet can also be modified during a debugging session to suit specific applications.

### 3.2 The Nub Interface

The nub interface is designed to be as small as possible while supporting the fundamental debugging operations common to all debuggers [6]. Figure 5 summarizes the complete API. The nub does not support high-level facilities, such as expression evaluation or specific symbol-table formats, because these facilities can be implemented by other interfaces or by debuggers themselves.

`.Nub_set` and `.Nub_remove` set and remove breakpoints, which are specified by a file name, line number, and character position. Unlike most debuggers, breakpoints specify the locations of expressions, not lines. So, for example, it is possible to set a breakpoint on the increment part of a C for loop.

`.Nub_src` accepts incomplete breakpoints, in which any of the file name, line number, or character position are omitted, and invokes a debugger callback on all possible breakpoints that “match” the incomplete one. deet

|                          |  |
|--------------------------|--|
| <code>_Nub.init</code>   | initialize the nub                     |
| <code>_Nub.set</code>    | set a breakpoint                       |
| <code>_Nub.remove</code> | remove a breakpoint                    |
| <code>_Nub.src</code>    | visit breakpoints with a given pattern |
| <code>_Nub.frame</code>  | move to a specific stack frame         |
| <code>_Nub.fetch</code>  | read the target's memory               |
| <code>_Nub.store</code>  | write the target's memory              |

Figure 5: The nub interface

uses this function to determine which breakpoint to set when a user clicks on a line.

`_Nub.fetch` and `_Nub.store` access the target's memory. They accept a buffer address, a byte count, and an address space identifier, and read/write data from/to the target. The address space identifier may specify an operating-system address space, such as the text or code segments. It can also specify logical address spaces that may not be part of the target, like the symbol table, for example. It is the nub's responsibility to access the appropriate data. Debuggers can view all data about the target as if they were stored in memory.

`lcc` emits machine-independent symbol tables in the target's address space, and `_Nub.fetch` reads these data. Another `cdb`-specific, but machine-independent, interface provides a higher-level view of the C symbol table as an inverted tree of symbol objects. If we were using the nub with, say, `Modula-3`, this high-level interface would have to be replaced with one specific to `Modula-3`, and that interface would use `_Nub.fetch` to read the symbol table generated by the `Modula-3` compiler. There's nothing special about `lcc`; other C compilers could be used given an appropriate nub implementation. It's the nub that's the critical component, not the compiler.

### 3.3 The `deet` Nub Interface

`deet` includes versions of the nub and symbol-table functions for use with `Tcl` or `tksh`. These `tksh` commands differ from the C routines in two ways: They are at a higher level, because they manipulate source-level symbols, types, and values, and they accept and return strings, so that they can be used in `Tcl` or `tksh` scripts. The complete list appears in Figure 6.

`deet.breakpoint` is a combination of `_Nub.set`, `_Nub.remove`, and `_Nub.src`. The `-set` option sets all of the given breakpoints, which might be incomplete; that is, *file* can be "", and *line* and *character* can be zero. The `-delete` option removes breakpoints, and the `-list` option lists possible breakpoints.

`deet.frame` is equivalent to `_Nub.frame`: With no

arguments, it returns the current frame as a `Tcl` list containing the frame number, the function name, and a file, line number, character number triple that gives location of execution within that frame. With an integer argument *n*, `deet.frame` makes frame *n* the current frame and returns the null string. Frames are numbered from the top of the stack, beginning with zero.

`deet.getval` and `deet.putval` commands are similar to `_Nub.fetch` and `_Nub.store`, but require type information to be specified along with the value, because `Tcl` deals only with strings. `Tcl` cannot, for example, deal directly with binary floating-point values or with structures. Types are specified by type identifiers, which are just generated strings. `deet.getval` returns a string representation for the value of *type* at *address*, and `deet.putval` writes the *value* of *type* to locations beginning at *address*.

`deet.sym` and `deet.type` return symbol table data. A symbol-table entry is a `Tcl` list { *name*, *type*, *address* }. `deet.sym`'s `-all` option returns a list of all of the symbols in the target; that is, a list of three-element lists. The `-files` option returns a list of all of the source files in the target. The `-locals` and `-params` options return lists of the locals and parameters for the current frame. The `-name name` returns the symbol-table entry for *name*, or an error if *name* is not a visible symbol.

`deet.type` returns a string describing the type represented by the identifier *type*. If *type* represents `int`, `deet.type` returns "int". Similarly, if *type* represents `T *`, `E [n]`, or a structure type, `deet.type` returns, respectively, the type identifier for *T*, *n* and the type identifier for *E*, and a list of names and type identifiers for the fields.

`NeD` [10] is another debugger built on a set of debugging primitives. This set is larger than the set of nub functions and the `NeD` primitives are at a somewhat higher level. `NeD`'s primitives are written in `Tcl` extended with a set of debugging functions. While these functions present a nearly platform-independent interface, their implementation appears to be platform-dependent and perhaps nontrivial. Also, `NeD` has no user interface per se; it uses `Tcl` in the same way as `deet`

|  |   |
|--|---|
| deet_open  | initialize the target                                   |
| deet_breakpoint { -set   -delete   -list }                         | <i>file line character</i>                              |
|  | set, remove, and list breakpoints                       |
| deet_frame [ <i>n</i> ]  | get/set current frame                                   |
| deet_getval <i>type address</i>                                    | read a value of <i>type</i> from <i>address</i>         |
| deet_putval <i>type address value</i>                              | write the <i>value</i> of <i>type</i> to <i>address</i> |
| deet_continue  | resume execution  |
| deet_sym { -all   -files   -locals   -params   -name <i>name</i> } | finds the symbol-table entries                          |
| deet_type <i>type</i>  | get <i>symbol</i> 's type information                   |

Figure 6: deet's nub interface

uses the nub functions, while deet uses Tel as its user-interface language, as illustrated in the next section.

## 4 Programming in deet

Much of deet itself is written in Tcl and tksh, using the deet.\* nub commands described above. Users can extend deet by writing Tel and tksh commands; for example, features like conditional breakpoints and non-trivial program queries can be written in tksh. deet can also be extended by external programs. This section illustrates some typical extensions.

Simple extensions can be written directly in tksh. For example, the following script displays all of the null elements in an array, the name of which is supplied as an argument.

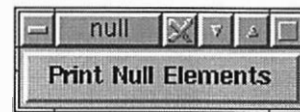
```
function nullElements {
    typeset arr=$1
    integer i s=$(arraySize $arr)
    for (( i=0 ; i < s ; i++ )); do
        if [[ $(var "$arr[$i]") == 0x0 ]]
        then
            print "Element $arr[$i] null"
        fi
    done
}
```

nullElements uses two external tksh functions: arraySize, which returns the number of elements in an array, and var, which returns the value of a variable. These functions are provided as part of deet. The for loop visits each element of the array specified by the first argument, retrieves its value, and prints the array name and index of the null elements.

User-defined functions can also manipulate deet's interface. For example, if we're checking repeatedly for null elements in hashtable, we can construct a button to do the job in one click:

```
toplevel .null
pack $(button .null.b \
    -text "Print Null Elements" \
    -command "nullElements hashtable")
```

This code builds the button:



Tel scripts can be invoked with the source command, which is a tksh built-in. source uses the Tel parser to parse its input, and uses tksh variables and functions in variable and command substitutions. Here's a simple example:

```
function foo {
    X=37
    print "${bar test}"
}

source <<'EOT'
proc bar {args} {
    global X
    set X [expr $X + 1]
    return "bar: args: $args, X: $X"
}
EOT
```

A call to foo prints

```
bar: args: test, X: 38
```

Note that the Tel procedure bar can use and modify the shell variable X. Tel source code can also invoke tksh functions and built-ins.

deet's name space is separate from the target's name space. Accessing a target variable from a tksh script requires a special function, var, which uses the target's symbol table to lookup the variable name and retrieve its

value. `var` is sufficient for one-shot lookups, but it's tedious for repeated uses of specific target variables. For these uses, `deet` provides `linkvar name`, which creates a new shell variable that is essentially an alias for the target variable `name`. `linkvar` is implemented with *discipline functions*, which are similar to trapped variables in SNOBOL4 [5]. A discipline function is a shell function that is associated with a variable, and that function is invoked whenever the variable is read or written. Thus, associating the function

```
function foo.get {
    foo="$ (var foo) "
}
```

with `foo` arranges for the target variable to be fetched every time the shell variable `foo` is read.

`deet`'s capabilities are easily extended by writing Tcl and tksh scripts that use the built-in debugger commands. An important advantage using a shell as the debugging language is that the shell can use *any* external tool. For example, it's relatively easy to extend `deet` to display linked data structures graphically as directed graphs. This feature is similar to that provided by the Data Display Debugger (ddd) [16], but the implementation is much simpler, because `deet` uses existing tools instead of building its own facilities. `deet` runs `dotty`, a program for drawing directed graphs [9], to draw the graph, sending it the appropriate input for the data structure of interest. Figure 7 shows an example of `dotty`'s output. The tksh script that invokes `dotty` is only about 60 lines of code, and it handles any linked data structure.

## 5 Implementation

`deet` is written in tksh; each `deet` command is implemented as one or more tksh functions that call the built-in Tcl nub commands. An example is the `b` function shown in Figure 8, which uses `deet.breakpoint` to set breakpoints. The size of this function is as important as its details: most debugging features are easily implemented in tens of lines of tksh code.

`b` begins by converting its first argument into file, line number, and character number values. When an incomplete breakpoint is specified, some of these values will be converted to null values. For example, `b 8` causes the `cvtbp` function for the argument 8 to become the value of line and for file and char to be null. Next, `b` invokes `deet.breakpoint -list` to list all breakpoints matching the incomplete breakpoint. If there is more than one match, a list of possible breakpoints is displayed and no breakpoints are set. If there are no

matches, a diagnostic is issued. Finally, if there is exactly one match, that breakpoint is set. The associative array `breakpoint` keeps track of the set breakpoints. The nub doesn't keep track of breakpoints because it is designed to do as little as possible. If the second argument specifies a condition for the breakpoint, it's stored as the value for the breakpoint array entry. Finally, the source window (if it exists) is updated to highlight the set breakpoint.

The nub interface can set and remove breakpoints, but it cannot single-step the target [6]. `deet`'s `step` function implements single-stepping by setting and removing breakpoints:

```
function step {
    if [[ $cdbMode != "step" ]]; then
        deet_breakpoint -set "" 0 0
    fi
    cdbMode=step
    cdbgo          # resume execution
}
```

Calling `deet.breakpoint` with null values for the file, line number, and character number sets every breakpoint. Implemented naively, setting every breakpoint is expensive in large programs. But the nub could recognize this special case and use a more efficient implementation. As described in Section 6.2, our implementation of the Tcl nub functions on top of `gdb` exploits this possibility.

## 6 Replacing the Nub

An important aspect of `deet`'s "piece-parts" design is that superior replacements could be used for each part without disturbing the others. For example, a more efficient, machine-specific nub could be used in place of `cdb`'s machine-independent nub; or a better or more familiar user interface could be used.

To demonstrate this flexibility, we've implemented three alternative versions of `deet`'s pieces: a version of the nub for Java [1], a nub that works by communicating with `gdb`, and a replacement for the user-interface component that emulates `gdb`'s command-line interface. These limited experiments also reveal strengths and weaknesses in the nub-based design. If `gdb` cannot emulate the nub, for example, then a simple nub offers facilities beyond those of some popular debuggers. If the nub cannot support `gdb`, then the nub is missing some important facilities.

### 6.1 A Nub for Java

The Java Developer's Kit contains a debugging package (a set of classes) that can be used to explore and con-

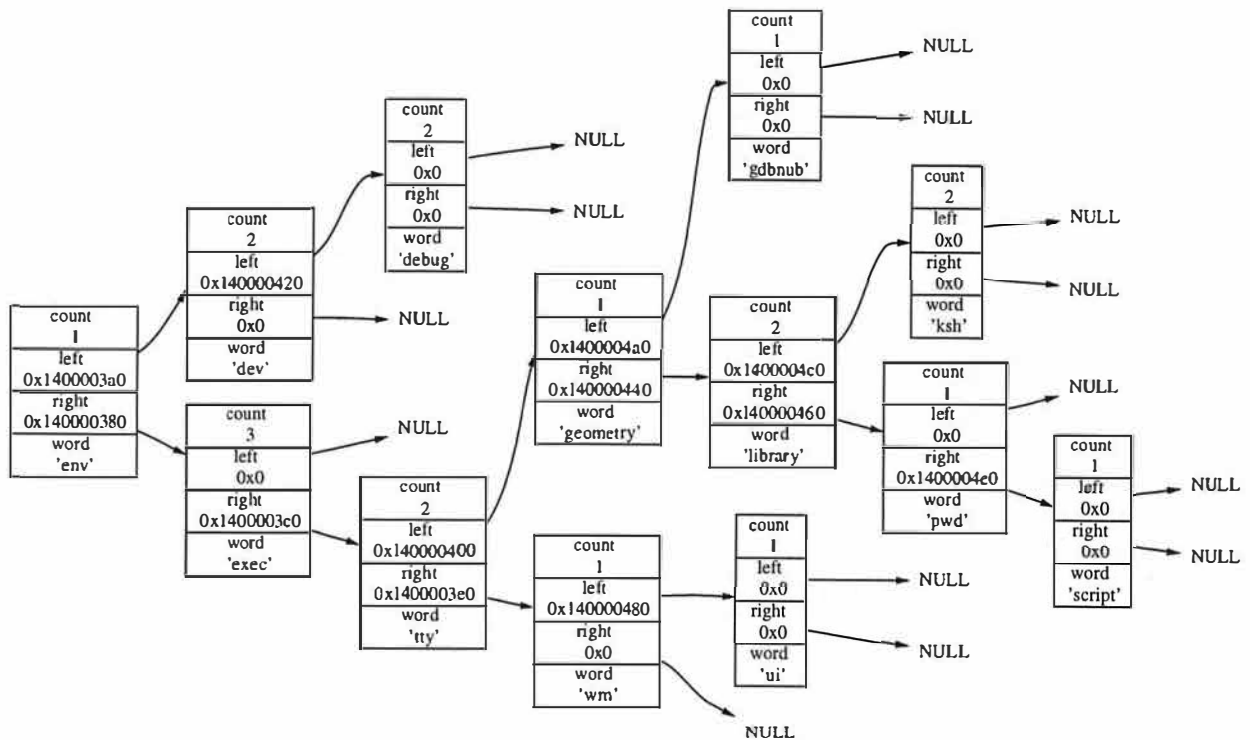


Figure 7: Tree generated with deet and dotty

```

function b { # breakpoint [action]
  integer char=0 line=0
  typeset file point="$1" action="${2-':'}" msg
  eval set -- $(cvtbp "$point")
  file="$1" line="$2" char="$3"
  typeset bp="$(deet_breakpoint -list "$file" "$line" "$char")"
  eval set -- $bp
  if (( $# > 1 )); then
    msg="Pick one of $bp"
  elif (( $# < 1 )); then
    msg="No breakpoint in on line $line char $char"
  else
    deet_breakpoint -set "$file" "$line" "$char"
    set -- $1
    breakpoint["$1:$2.$3"]="$action"
    [[ $CdbWindow ]] && TextDispBpOn $2
  fi
  [[ $msg ]] && print -- "$msg"
}

```

Figure 8: Implementation of the b command

trol the state of a target Java program. These classes are designed to support a variety of Java debuggers. Java comes with `jdb`, a simple command-line debugger, that is implemented with the debugging package, and this package is intended to be used to write more sophisticated graphical debuggers. The package works by spawning an instance of the Java runtime with the target and communicating with it via message passing.

Although the Java nub itself could spawn the runtime and the target, it's simpler to use the debugging package. Implementing a nub for Java required writing the nub interface in terms of Java's debugging methods, which takes only a couple hundred lines of Java. The routines in the nub read messages from a socket, process them using the methods in the Java debugger package, and write the result messages back to the socket. A central method reads messages, decodes them, and calls the appropriate methods for each message.

Figure 9 shows the method `frameCmd`, which performs the same task as `deet.frame`. When called with an argument, `frameCmd` sets the current frame to the number specified by the argument by finding the current frame with the `getCurrentFrameIndex` method and calling the up and down methods as needed. If there is no argument, `frameCmd` returns information about the current frame. This information is returned by calling methods in the debugging classes `RemoteThread`, `RemoteStackFrame`, and `RemoteClass`.

Thus, with a couple hundred lines of Java code, `deet` can be used to debug Java programs with the same set of features that are used to debug C. Unfortunately, the nub interface does not currently support threads, which limits the usefulness of the Java debugger.

A similar approach can probably be used on any system that has a debugging interface. For example, `deet` can be ported to Windows by implementing a nub in terms of Microsoft's debugging API.

## 6.2 Using `gdb` as a Nub

We've used `gdb` to build a variant of `deet` that uses `gdb` as the nub, and a variant that uses `gdb` as the user interface.

Figure 10 shows how `gdb` replaces the nub. `gdbnub` communicates with `gdb`, which runs as a separate process. `gdbnub` translates nub function calls into `gdb` commands, sends these commands to `gdb`, and parses `gdb`'s responses. Another approach would have been to modify `gdb`'s code, but past experience shows that modifying `gdb` is a painstaking process [4]. Using `tksh` makes the approach illustrated in Figure 10 *much* simpler: the implementation takes only about 500 lines.

The only nub feature that was not possible to implement with `gdb` was setting breakpoints on any expres-

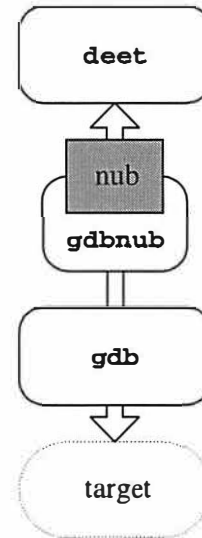


Figure 10: Emulating the nub with `gdb`

sion. Some nub functions were relatively easy to implement, but not efficiently. For example, `gdb` doesn't provide a way to list all of the possible breakpoints in a file. This was implemented by attempting to set a breakpoint at every line in each file, and checking which breakpoints were successfully set. Fortunately, listing all of the possible breakpoints is rarely done; listing the breakpoints in a specific line is the common usage.

As described at the end of Section 5, `deet` steps through a program by issuing the nub command to set every breakpoint in the target, which is very inefficient. `gdbnub` takes advantage of `gdb`'s single-stepping feature by using the `deet.breakpoint` implementation shown in Figure 11. When null values are passed to the `-set` option, a variable is set that puts the `gdbnub` into "stepping" mode when execution is resumed. Similarly, when null values are supplied with the `-delete` option, `gdbnub` reverts to "continuation" mode, and all temporary breakpoints are removed.

When `gdb` is used as the nub, `deet` can be thought of as a graphical front end to `gdb`. It provides facilities similar to `ddd`'s, which is also a front end for `gdb`. However, `ddd` is as not programmable.

## 6.3 A Nub for `gdb`

Implementing `gdb` on top of the nub is difficult, because `gdb` is a huge program and has a large number of features. Some of the features in `gdb` are inherently absent in the nub. For instance, `gdb` allows the target to be examined at the machine level; `gdb` can examine registers and single-step instructions. The nub interface is machine-independent, so it cannot provide these

```

public void frameCmd(RemoteThread t, String args[]) throws Exception {
    if (args.length == 2) {
        int oldFrame = t.getCurrentFrameIndex();
        int newFrame = Integer.parseInt(args[1]);
        try {
            if (oldFrame < newFrame)
                t.up(newFrame-oldFrame);
            else if (oldFrame > newFrame)
                t.down(oldFrame-newFrame);
        } catch (ArrayIndexOutOfBoundsException e) {
            outputError();
        }
    } else {
        RemoteStackFrame s = t.getCurrentFrame();
        RemoteClass c = s.getRemoteClass();
        outputItem(t.getCurrentFrameIndex());
        outputItem(c.getName() + "." + s.getMethodName());
        outputItem(c.getSourceFileName());
        outputItem(s.getLineNumber());
        outputItem("0"); /* No support for char position */
    }
}
}

```

Figure 9: Implementation of deet\_frame for Java

```

function deet_breakpoint {
    typeset i action=list
    case $1 in
        -l*) shift ;;
        -s*) action=set ; shift
            if [[ $1 = "" && $2 = "0" && $3 = "0" ]]; then
                sendCommand "b main"
                CONT_CMD="step"
                return 0
            fi ;;
        -d*) action=delete ; shift
            if [[ $1 = "" && $2 = "0" && $3 = "0" ]]; then
                sendCommand "delete"
                for i in "${!GdbBreakpoint[@]}"
                do unset GdbBreakpoint[$i]
                done
                CONT_CMD="cont"
                return 0
            fi ;;
    esac
}

```

Figure 11: Implementing single-stepping

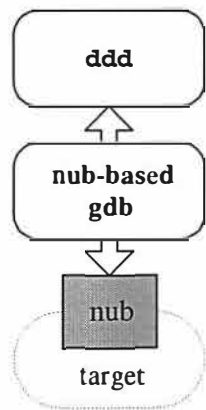


Figure 12: Running gdb with the nub

machine-dependent features. Similar caveats apply to data watchpoints, which gdb supports on machines with the appropriate hardware. gdb also supports features that are irrelevant to debugging, per se, such as controlling terminal modes and displaying gdb-specific online help.

Our third experiment thus focuses on only those gdb features used by a graphical front end, like ddd. Figure 12 shows the organization of this experiment.

The implementation of gdb using the nub was written with deet commands. gdb's `frame` command illustrates the general approach. `frame` controls the current frame in the stack of the target. Invoking `frame` with an argument directs gdb's attention to a specific frame. If no argument is specified, the current function with its arguments and location is displayed. For example:

```
(gdb) frame
#0  lookup (word=0x11ffff8e0 "a",
          p=0x140000010) at test/lookup.c:15
```

Figure 13 shows the tksh implementation of `frame`. The function uses `deet_frame` to move the nub's attention to a new frame. It also uses `deet_sym` and `deet_getval` to fetch and display the frame's parameters and their values.

This implementation of gdb, although incomplete, is only around 1,000 lines of ksh. It implements enough features to support ddd. gdb features that were not implemented include:

- Debugging a target that is already running, which gdb can on machines where this is possible.
- Invoking target functions from the debugger; the nub doesn't support this feature, because a separate evaluation facility can support it [13].

- Examining core dumps; this feature could be supported by writing a nub specifically for browsing core dumps.
- Interrupting a running target.
- Handling signals.

## 7 Discussion

deet's front end runs on any machine on which tksh runs, which currently includes virtually all UNIX variants, Windows NT and Windows 95. Graphical debuggers that work consistently under both UNIX and Windows are scarce, and having a uniform interface can be important. Programmers writing code for multiple platforms can debug applications without having to learn multiple environments. Another advantage of a uniform interface is that one set of debugging scripts is often sufficient for all platforms.

The nub hides most of the difficult portability issues. deet is available on all of gcc's platforms, because its nub interface is machine-independent. deet is also available on platforms that support gdb, because it can use the nub that runs on top of gdb. deet can be made available on other platforms by writing a new, platform-specific nub. Typical nub implementations take less than a thousand lines of code, so they aren't trivial, but the effort required is tiny compared to porting a machine-specific debugger.

deet demonstrates that it is possible to build a usable debugger with a graphical user interface from simpler components, and, as the dotted example illustrates, that the result is more than just the sum of the parts. deet also confirms cdb's premise that most of a debugger is machine-independent, and that the fundamental machine-specific debugging facilities can be encapsulated in a small, machine-independent nub interface.

deet doesn't have all of the features offered by PC debuggers and UNIX debuggers like gdb. But it does provide the most important ones—at a fraction of the implementation cost. deet is about 1,500 lines of tksh code, and the machine-independent nub and related compiler support (in gcc) total around 800 lines of C. These 2,300 lines of code are orders of magnitude smaller than gdb's 150,000 lines and ddd's 90,000 lines.

Programmers interact with most debuggers in the target's source language plus a few debugger-specific commands. For example, programmers throw C expressions at gdb to browse the state of a buggy target. The advantage of this approach is that programmers don't have to learn another language to use the debugger. But, as Acid [15] and Duel [4] demonstrate, exploring a program's state is fundamentally different than writing the

```

function frame { # [num]
  [[ $1 != "" ]] && deet_frame $1 2> /dev/null
  set -- $(deet_frame)
  typeset num=$1 name=$2 file=$3 line=$4 char=$5
  typeset params="$(deet_sym -params)" p result
  result="#$num $name("
  eval set -A parm $params
  for p in "${parm[@]"; do
    set -- $p
    result="$result${1##*:}=$(deet_getval "$2" "$3"), "
  done
  print -- "${result%', ')} at $file:$line"
}

```

Figure 13: ksh implementation of gdb's frame command

program in the first place, and this exploration can be done much more effectively in a higher-level language. deet also supports this view; Tcl and tksh seem to be better languages for writing debugging code than languages like C and C++. Similar comments may apply to other high-level scripting languages, like Perl.

## References

- [1] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, Reading, MA, 1996.
- [2] M. Bolsky and D. Korn. *The New KornShell Command and Programming Language*. Prentice Hall, Upper Saddle River, NJ, second edition, 1995.
- [3] C. W. Fraser and D. R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley, Menlo Park, CA, 1995.
- [4] M. Golan and D. R. Hanson. DUEL — a very high-level debugging language. In *Proceedings of the Winter USENIX Technical Conference*, pages 107–117, San Diego, CA, Jan. 1993.
- [5] D. R. Hanson. Variable associations in SNOBOL4. *Software—Practice and Experience*, 6(2):245–254, Apr. 1976.
- [6] D. R. Hanson and M. Raghavachari. A machine-independent debugger. *Software—Practice and Experience*, 26(11):1277–1299, Nov. 1996.
- [7] D. G. Korn. ksh: An extensible high level language. In *Proceedings of the Very High Level Languages Symposium (VHLL)*, pages 129–146, Santa Fe, NM, October 1994.
- [8] J. L. Korn. Tksh: A Tcl library for KornShell. In *Proceedings of the USENIX Tcl/Tk Workshop*, pages 149–159, Monterey, CA, July 1996.
- [9] E. Koutsofios and S. C. North. Applications of graph visualization. In *Proceedings of Graphics Interface 1994 Conference*, pages 235–245, Banff, Canada, May 1994.
- [10] P. Maybee. NeD: The network extensible debugger. In *Proceedings of the Winter USENIX Technical Conference*, pages 145–153, San Antonio, TX, July 1992.
- [11] Microsoft Corp., Redmond, WA. *Microsoft Visual C++, Reference Volume II*, 1993.
- [12] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, MA, 1994.
- [13] N. Ramsey and D. R. Hanson. A retargetable debugger. *Proceedings of the SIGPLAN'92 Conference on Programming Language Design and Implementation, SIGPLAN Notices*, 27(7):22–31, July 1992.
- [14] R. M. Stallman and R. H. Pesch. Using GDB: A guide to the GNU source-level debugger, GDB version 4.0. Technical report, Free Software Foundation, Cambridge, MA, July 1991.
- [15] P. Winterbottom. Acid: A debugger built from a language. In *Proceedings of the Winter USENIX Technical Conference*, pages 211–222, San Francisco, CA, Jan. 1994.
- [16] A. Zeller and D. Lütkehaus. DDD — a free graphical front-end for UNIX debuggers. *SIGPLAN Notices*, 31(1):22–27, January 1996.

# Cget, Cput, and Stage - Safe File Transport Tools for the Internet

Bill Cheswick  
*Bell Laboratories, Lucent Technologies*  
ches@bell-labs.com

## Abstract

*Cget*, *cput*, and *stage* are three simple programs that implement authenticated or encrypted file transfers on the Internet. *Cget* and *cput* read and write files to a remote host, and *stage* ensures that a remote directory accurately mirrors a local master directory.

These routines use private key cryptography for authentication and privacy between pairs of secured hosts. They are simple, paranoid Unix tools that can be used to support systems that operate in a hostile environment.

## 1 Introduction

A host can be made reasonably resistant to compromise from the Internet if it isn't running any dangerous network services. Most hosts don't come from the manufacturer configured in this manner—they have to be stripped of all of their network services by hand. Then only the desired services are installed. If the network services are secure (a big "if"), then machines are much harder to breach.

Two such secure hosts can exchange files and remain reasonably secure if there is a safe file transport service available. This file transfer service would have to be resistant to all the popular attacks found on the Internet today, including the recent IP spoofing[12] and TCP hijacking[10]. If the data is sensitive, then the service would also have to ensure privacy.

Such a service is needed often these days. In particular, publicly-available Internet services like FTP and http are often provided by hosts running on the dirty side of the firewall, usually in a DMZ. How can we administer such hosts, perhaps from the relative safety of another host behind a firewall? How do we install new programs, or install new content?

The standard out-of-the-box network services do not provide this security. In fact, they have had a history of jeopardizing their servers. FTP has been a constant source of trouble: its passwords are easy to sniff, its protocol has various flaws, and various servers have had security holes [5, CA-88:01, CA-92:09, CA-93:06, CA-94:07, CA-94:08, CA-95:16]. *Rcp* relies on address-based authentication, and the

integrity of the Domain Name System, which is easy to fool [4, 16]. It is also susceptible to IP spoofing attacks [14]. NFS has a variety of weaknesses: it can be fooled with address spoofing, root handles can be sniffed or guessed, and it relies on RPC services that have security weaknesses of their own.

These services are frequent targets for successful hackers. Still, they are often employed because they are widely available, and most developers are familiar with them—even when they are clearly unsuited for the job. Developers don't have time to build new tools: the frenzy of Internet hype and growth can leave management focused on time-to-market issues. Security is often left to the last minute, and patched in after the design is finished.

Marcus Ranum and I faced these problems in the fall of 1995. We encountered *ad hoc* solutions using standard, dangerous Internet services. For example, billing data was transferred in the clear using FTP. Developers cast about for ways to transfer configuration files and other important data. The standard tools they used were jeopardizing some very important hosts.

We wanted a very simple solution, in the tradition of small Unix tools. This is not a very tall order: it requires a simple file transfer program running some strong cryptographic or authentication routines, and a shared secret key.

We were only setting up a few point-to-point links, so it was easy to distribute a secret key to each end of a connection. We didn't want an authentication server or need public key cryptography. We envisioned that a pair of simple programs, plus

a key file and a configuration file, were all that were needed. A simple implementation meant that more people were likely to understand and use the software, even if they were in a hurry to make a deadline.

There were a variety of possible off-the-shelf solutions available at the time: our problems were not new. Most of them appeared to have adequate security, but none were as simple as we desired. (Some of these are discussed in section 9.)

We have a better chance of avoiding security bugs if the programs are small and simple.

Marcus and I built three programs. Each was as simple as possible, and written with all the minimalism and paranoia we could muster. *Cget* reads a single file from a remote server, and *cput* writes a file back. *Cget* and *cput* are quite primitive: they do not support file deletion, directory creation, or program execution. (They were originally named *get* and *put*, but that clashed with SCCS's routines of the same name.) *Stage* mirrors a master directory to a slave host. Files and directories are created, deleted, and downloaded as needed. It is ideal for updating an external web server or FTP archive from an internal staging host. *Unstage*, a recent addition, works in reverse, updating a local copy of a remote master directory. It can be used to suck logs from a server, or perhaps get a distribution from a master tree.

Our initial cryptographic routines used DES encryption. For many applications, this is overkill. Most transfers are not secret—FTP and Web data are generally intended for public viewing.

If we kept strong encryption, it would make the software release difficult and unlikely.

So the crypto layer has been rewritten—the interface and code are cleaner than the first version. The new crypto routines use only HMAC keyed message digests to protect the conversation. If my authentication protocol is OK (always a big ‘if’), an eavesdropper may watch or interrupt a session, but cannot modifier replay a session without detection.

The next section describes the authentication protocol and some cryptographic issues. Section 3 describes the user interface to the cryptographic protocols. Server design issues are explored in section 4. The *stage* service is discussed in some detail in Section 5. Section 6 has a couple of applications for these programs, including the confinement of an arbitrary TCP service. Section 7 covers vulnerabilities, and section 8 has some performance figures. A tiny sample of the related work is discussed in section 9. Section 10 describes some enhancements and limitations to these routines, and availability information is in section 11. Appendix A describes the

staging protocol.

## 2 The Authentication Protocol

The client and server exchange messages in SSL format, although we do not use SSL's complex key setup. Each message contains a two byte length field, the payload, and a 16 byte binary digest.

I use the HMAC[3] digest with MD5, which appears to be headed for general usage on the Internet. An HMAC digest of a message *M* using key *k* is shown as

$$[M]_k$$

The client and server share a secret key,  $K_{ss}$ . The protocol uses challenges in both directions to derive session keys. The challenges are sixteen random bytes encoded in pairs of hex digits. The session key for each end is derived from  $K_{ss}$  and the challenge from the other end:

$$K_s : [C_c]_{K_{ss}}$$

$$K_c : [C_s]_{K_{ss}}$$

The server writes with  $K_s$ , the client with  $K_c$ . The initial exchange between client *C* and server *S* proceeds as follows:

Message 1  $C \rightarrow S : N, C_c, [N, C_c, S_c]_0$

Message 2  $S \rightarrow C : C_s, [C_s, S_s]_{K_s}$

Message 3  $C \rightarrow S : \text{"OK"}, [\text{"OK"}, S_c]_{K_c}$

Here *N* is a service name (see section 4.1), and  $S_c$  and  $S_s$  are sequence numbers,  $C_c$  is the client's challenge and  $C_s$  is the server's challenge. The sequence numbers are four bytes long.  $S_c$  starts at zero, and  $S_s$  starts at  $2^{31}$ .

Message 1 delivers the client's challenge. It uses the key 0, which helps us detect casual probes of the service. Message 2 proves to the client that the server is using the new challenge, has the secret key, and provides the server's challenge. Message 3 has a trivial payload, but proves to the server that the client is using the fresh challenge and the secret key.

The session keys are used to prevent replay attacks using messages from previous sessions. If we simply keyed our digests with the  $K_{ss}$ , an attacker could replay a previous session, perhaps replacing a new file with some older one. Each end uses a different session key so a message can't be played back to its originator. Similarly, the sequence numbers prevent replays of earlier messages in the same session. Without these, an attacker might hijack the TCP session, and replay earlier messages. The sequence numbers differentiate the hashes from each end of the conversation.

Although the client can force the server to use a specific challenge, and therefore the same key, it can't finish the protocol initialization without using the server's fresh challenge and the secret key.

A man-in-the-middle can't change a message without detection: he cannot determine the session key without the secret key, and he cannot obtain the right answers from an additional connection to the server, since the session key will be different.

This is a simple protocol, and it looks like it ought to do the job, but I am not a cryptographer, and history teaches that it is hard to get cryptographic protocols right. Though I don't see how this protocol can be abused, I'd feel better if each session key were based on both challenges.

## 2.1 Administrative concerns

Protocol setup can interact with administrative concerns. Our original protocol was terse and unhelpful when, say, one end had the wrong key. The obscurity may have added some security, but it sure didn't help our users who were trying to set up the service.

The setup may fail for a number of reasons: the key is wrong or missing, the key file is not readable, the service is non-existent, etc. Each of these errors means that the server cannot return the correct digest in its first message. If the digest is wrong during protocol setup, the client checks the payload for the string "remote reported an error:". If present, the rest of the message is an error message from the server describing the problem.

Until the exchange is complete, the protocol is subject to attack. In particular, it is possible for an attacker to inject a false error message during this setup phase.

## 2.2 Keys

Secret keys are appropriate here: we don't need public key cryptography. The usual complaint about secret keys is the distribution problem—how do we move them around securely? For us, these programs are only employed in a handful of hosts, involving perhaps a dozen services and their keys.

Our keys are printed in hex bytes or base 64 encoding, both human-readable. They can be distributed by hand or over the phone when the service is installed. We type ours in at the consoles of the hosts involved.

This might not scale to a large setup, but one could imagine an ISP allowing a thousand customers to use *stage* to update a thousand separate web directories. It would not be much harder to distribute

a binary key than a password that a user has to remember, and the client wouldn't need an account on the web server. (This is always a good feature: users are annoying, and tend to disrupt security arrangements.)

The secret keys are generated by a program named *makekey*. Its keys, and the protocol's session challenges, are generated with *truerand*[11]. (*Truerand* runs a counter in a tight CPU loop while waiting for an alarm timeout some milliseconds later. The bottom two or three bits of the counter are considered random.)

We use full random binary keys: there are no passwords that a user must remember.

## 3 Interface Routines

To use the crypto routines, the client uses the following code:

```
fd = tcpconnect(host, port);
ep = start_client_crypto(fd, key,
                        srv_nam);
if (ep != 0) {
    /* error */
}
n = cread(fd, buf, sizeof(buf));
n = cwrite(fd, &gt, ngt);
if (n < 0) {
    perror("writing gunilla table");
    exit(1);
}
```

and the server uses

```
fd = bindto(service_port);
ep = start_server_crypto(fd);
if (ep) {
    perror(ep);
    exit(1);
}
n = cwrite(fd, "hello", 6);
n = cread(fd, buf, sizeof(buf));
...
```

(Some error processing is simplified for clarity.) *cread* and *cwrite* are analogous to the standard I/O routines. *Cperror* reports standard or cryptographic errors. This protocol preserves message delimiters: each *cread* will return only the bytes sent by the corresponding write. The maximum message size is  $2^{16} - 1$ .

The server must supply a routine named *setservice*, which obtains the secret key or returns an error message.

## 4 Services, servers, and server trust

Our server programs assume that they have no friends. For example, they shed privilege early to minimize the code we must trust. The server programs obtain the key for the calling host, *chroot* to a target directory, and change their user id and group to some less-privileged account. All inputs from the client are carefully checked for pathological values.

We have two servers: *getd* for *cget* and *cput*, and *staged* for *stage* and *unstage*. They operate on different TCP ports, and are called by *inetd*.

### 4.1 Service names

Originally, these routines were keyed to a host's numeric IP address. A single client host could *cget/cput* to one area of a server, and *stage* to another. If others needed to *stage* to the same server, they would have to connect from a different client host. The IP address of the caller was used to select the proper key and service configuration on the server. Though it provided only slight security, the connection had to originate from that IP address. The secret key provides the real security.

This approach worked well for simple setups, but the one-service-per-client limitation became inconvenient as the services were used more. Also, a traveling host couldn't access a fixed server, because the client's IP address was unpredictable. We could provide additional services on different ports on the server, but this is awkward.

The new authentication protocol includes a service name. The key and other information are based on this name. A single client host can have a number of services on a given server. Different users can have different access to the same server, all controlled at the client end by read access to the relevant key file. This trust model shouldn't be pushed too far: Unix *root* accounts are generally not very resistant to user attack. A serving host should extend about the same level of trust to all services from a given client.

### 4.2 Trusting the Server Software

So far, I have assumed that

1. we control the server machine entirely, and
2. we trust the server code until it drops privileges.

Neither assumption may be true if we would like to persuade some one else (say, an ISP) to run our servers on their host. They would be more willing to run our software if we don't need *root* permission,

and perhaps even more willing if they can contain our software with their own *chroot*.

The problem is that *chroot* requires *root* permission, and it is hard to change the user id safely with standard shell commands after the *chroot*. We have to include a *setuid* program within the software "jail" to change from user *root*, and the user with access to this directory might find a way to disable this program.

The *chroot* program needs an option to set the UID of the executing program. I wrote a trivial version of *chroot* named *jail* to do this. I can give this tiny program to the ISP. It's only a few lines long: they can examine it, trust it, and confine our servers nicely with it.

We have some problems when the server is enclosed in the jail. How does the program obtain its key, unless the key is stored within the jail itself? We may wish to keep the key secret from the user. If the key is stored in the jail, a remote user may have undesirable read access to it. It could be piped in through a file descriptor opened by *jail*, but that's a bit awkward to set up. The key could also be a parameter to the server program, but that can make it visible to other users on the serving host through the *ps* command.

It would also be nice to let the jailed server issue *syslog* messages. This requires possibly-dangerous special files inside the jail, or some mechanism for *jail* to perform the *openlog* and pass the file descriptor to the server in the jail.

I am not satisfied with the solutions to either of these problems. *Chroot* is a good start, but Unix lacks adequate confinement primitives.

## 5 Stage

Once the crypto routines were working for *cget* and *cput*, *stage* was an obvious application. *Stage* runs through a local master directory, comparing each file and directory with the contents of the remote slave directory. The master directory for a service is identified by an entry for that service in configuration file on the client, usually in */usr/local/etc/stage.conf*.

Like *cget/cput*, *stage* uses the service name to look up the appropriate key. *Staged* also uses the service name to determine the target directory, user id, and file permission mask. It uses *chroot* to confine itself to the target directory. This is good: it is somewhat more complex than *getd*, and therefore more likely to have bugs. It does check its input from the client carefully: strings can't be too long, "." is not allowed in path specifications, etc.

*Stage* will update a file if it has changed. A file is considered changed if its modification date or length are different, or if its MD5 checksum is different. The checksum is time consuming—an option suppresses this check. When a file is copied over, its modification date is set to match the master copy, if the operating system on the server allows it.

Ordinary users can use *stage* to update all or some portion of the master directory. It only takes a few seconds to check and transfer a few megabytes. *Stage* does not exit until the update is complete: there is no queuing mechanism involved. Should the program abort or fail for some reason, it can be rerun to ensure that the directories match.

The user can *stage* a file or a directory. Either must appear under the master directory. If he stages a non-existent path, that path will be deleted on the server, if it exists. Hence:

```
rm -rf foo
stage remote foo
```

will delete a directory or file named *foo* at the other end. There is a subtle distinction here:

```
stage remote *
```

and

```
stage remote .
```

are not the same. The first entry will update all existing files and directories in the current directory. The second will do that, plus delete any remote entries that don't appear locally.

*Stage* makes no special provisions for special files or soft or hard links. It makes no special provisions for files or directories that change during the staging process.

The user does not have to have read access to the key file: *stage* could be *setuid* to an account or group that has read permission for the key.

*Staged* serves *stage* requests on the remote server. It consults `/usr/local/etc/staged.conf`, which has one line for each supported service name. Each line contains the service name, the slave directory, and an optional UID and umask. File ownership is not propagated to the server: the slave files are owned by the account that the service is configured for. *Staged* logs all file activities via the *syslog* facility.

The *stage* client uses a little protocol to control the remote server. It is very simple (see Appendix A), a subset of basic file system access primitives. It could be optimized to improve performance.

## 6 Applications

These routines have been well-received, and are employed in a number of places in AT&T and Lucent. In our lab, *stage* has been providing quick and simple support for `ftp.research.att.com` for over a year, and now supports `ftp.research.bell-labs.com` and Lucent's Web service on `www.lucent.com`. We had to make a proxy version of *stage* for this last service to support it through our older firewalls. It replaced a clunky FTP-based implementation that was unable to delete files on the slave server. *Cput* has been especially useful for moving new binaries to external hosts.

One researcher has been transporting large, extremely sensitive databases over networks having dubious or unknown security. The encrypted version of *cput* is vital for him.

We have also supported our various external dirty Unix hosts with these tools. Automated jobs use *cput* to update mail alias files and internally-generated name server configuration files. *Cget* fetches daily log files so users can monitor access to the FTP directory without requiring logins or more invasive access to the server. Hal Purdy at AT&T Research has ported the encrypting *staged* to Windows 95 to support a roomful of PCs outside the firewall.

### 6.1 An Example: updating mail alias files

Traditional Unix file system permissions can be used to provide finer control over access to the target directory. For example, we have a slave mail directory on an external server that contains programs and data files. The master host is allowed to overwrite and create the mail alias files, but is not allowed to change the executable files in the slave directory. We could just split them into separate directories, but we are used to the current configuration.

The executable files are owned and writable by account *bin*. The directory and the alias files are owned by *daemon*. This lets the master host update and add new alias files, but it doesn't have write permission for the executable files. If we supplied complete Unix file system semantics, they could delete the executables (since they have write permission on the directory) and create new ones. But *cget* won't delete a file, only replace it. There's one additional protection: we can have the umask in *getd*'s configuration file clear the executable bits in downloaded files.

If we rely on UNIX's built-in file permissions,

we reduce the amount of special-case permission-checking software in *getd*. Smaller programs are safer.

## 6.2 Bob's TCP service

Bob has a service that he would like to offer the world. I don't know much about it. It looks at some files, answers queries on a TCP port, and writes logs. It doesn't need to be *root*, or use any fancy system services or files. It's just a program.

I like Bob, and I trust him (mostly). His service is not only harmless, it appears to be quite worthwhile. He'd like to run it on our external server.

I'd like to help, but I don't want to jeopardize a host that we've taken great pains to secure. I don't want an error on Bob's part to reduce that host's security. How far do we have to trust him?

We can lock Bob's software and files inside a *chroot* environment. He won't be running as *root*, and we won't leave any dangerous programs in his jail, so we are pretty confident that he can't get access to the rest of our file system. We add the following single line to */etc/inetd.conf*:

```
z3950 stream tcp nowait root
      /sbin/chroot chroot /usr/bob
      /bin/su - bob -c /tree/bin/zsrv
```

We use *chroot* to confine Bob, then *su* to give up *root* privilege, before Bob ever gets to execute an instruction. The security measures are right out there in the open, on the line where an auditor can see and understand them. Unfortunately, */bin/su* is inside his jail, so we use *jail* instead:

```
z3950 stream tcp nowait root
      /sbin/jail jail -u 99 -g 2 /usr/bob
      /tree/bin/zsrv
```

*Jail* is just like the *chroot* command with the additional *-u* and *-g* options to set the user id and group.

His program, *zsrv* should be linked with static libraries: it greatly simplifies the setup of the jail.

We split Bob's directory (*/usr/bob*) into two subdirectories: one (*/usr/bob/tree*) that he can stage into, and one (*/usr/bob/log*) that he can *cget* or *unstage* from. He can update his server software and other files under */tree* at will. He can change the network server, but it's always owned and executed on account *bob*, not *root*, so he is very unlikely to get out of his jail.

What can Bob, or someone who has hacked into Bob's account, do to us? Most of the problems are of the denial-of-service type:

- **File System Full.** He can fill the partition that holds */usr/bob/tree*. If this is a concern, we can put him in his own partition. Then it only breaks his program, though it may cause annoying log messages elsewhere.
- **Core dumps.** These can fall under the file-system-full problem. The *chroot* environment assures that the core dump will go in his directory, not somewhere else.
- **CPU hog.** If he uses too much of the CPU, we can *nice* him down before he starts.
- **Memory Full.** He could eat up or thrash memory.
- **Open Network Connections.** Our jail doesn't stop Bob from opening outgoing network connections. This could be abused in a few ways. In particular, he could try to embarrass us or take advantage of the good name of our serving host. Since these can be spoofed anyway, it shouldn't be an unusual problem.

Bob has been running his Z39.50 service for over a year. Aside from a few core dumps and some occasional configuration changes while companies lurch apart, Bob hasn't been a problem.

## 7 Vulnerabilities

The administrator is often the source of security problems. It is easy to leave key files around with unintended read permissions or incorrect ownership.

If the server is compromised, the game is lost: our original goal was to protect the server.

If the client is compromised, the intruder can gain access to all the services allowed to that host. The server's paranoia can block further spread of such an attack, limiting the intruder's access to a particular directory on the server. This can be an effective barrier which adds another layer to the depth of security.

If either host is compromised, then so are the keys, since they are stored on files. This alarms some people, but the keys are only as valuable as the host—once the host is compromised, the key is useless any way. We've attempted to limit the extent of such a catastrophe by limiting the trust each host has for the other.

The theft of a key can compromise the privacy of old sessions of the encrypting transport. We don't change keys often.

Session interruption can be a problem. If important security files are transmitted, the user must

note that an attacker can abort a transfer. For example, *tcpwrapper* [15] has a list of permitted connections in */etc/hosts.allow* and a stoplist in */etc/hosts.deny*. If they are transmitted in that order, there is a time when the */etc/hosts.allow* is in place without the exceptions. This time window can be enlarged by interfering with the second transfer. An administrator must bear these concerns in mind when moving security-related files. In this case, the */etc/hosts.deny* file should be transmitted first.

As mentioned before, the protocol's error messages may be subject to social engineering during setup. It's an unlikely attack since the timing is tight and the administrator is likely to be watching both the client and the server during initial configuration anyway.

As always, a network service accessible to the public can always receive denial-of-service attacks. Even if the digest is wrong, enough incoming packets can swamp any service, making it unavailable to its intended users.

## 8 Performance

We have found that these routines work with acceptable speed. With our first protocol using Eric Young's DES encryption library we could transfer 500 KB/s through the localhost port of an SGI running with 150 MHz CPUs.

In another test, our entire 700 megabyte FTP directory was copied between two fast SGI hosts through a firewall in about 47 minutes. When *stage* was rerun just after this transfer, without any updates, the check took about 330 CPU seconds and a little over 32 minutes. With checksumming turned off, it took six minutes.

We don't expect that an entire large directory will be staged more than about once a day. Users find the instant-update feature handy, and tend to stage the little bits they change quite often. The protocol is not especially efficient: it could be enhanced to speed up the file-checking phase.

## 9 Related Work

We have reinvented a well-rounded wheel. There are a number of software solutions available, some well-suited to their environments. We want a very light-weight solution. Private keys are fine, without authentication servers or fancy certificates. If the infrastructure is in place for such tools, use them: *Rcp* with Kerberos would be fine, if we were run-

ning Kerberos in the first place. But most of our customers don't run Kerberos.

*Ssh* [17] was fairly new when we did this work. It provides a transport protocol that will probably be quite secure—it is under repair at this writing. In late 1995, *ssh* was a beta release. But for our uses, *ssh* has too many features—it's too large. It offers optional features we don't want, like X11 transport and login facilities. We didn't necessarily want these additional services into our secure hosts. The code is more complicated, and there are more things to misconfigure. (We do use *ssh* in other places: it's a nice package.)

There are a number of mirroring and general software distribution packages available. The best known is *rdist* [9]. *Rdist* typically uses *rcp* or *ssh* for transport. The former is not appropriate, but *ssh* is a good choice for supporting *rdist*, and it has its enthusiastic supporters. There's a lot of mechanism there for our simpler applications. *Rdist* has earned three CERT advisories [6, 7, 8], which also makes us nervous.

Two other transport programs include *filetsf* [13] and *Mirror* [2], a Perl program. *Mirror* uses FTP for transport, *filetsf* uses *lpr/lpd*. Again, we don't want these additional services on our safe machines.

Other file transfer programs have appeared recently, such as *SSLftp*. See [18] for a wide assortment of related tools.

We've seen other batch approaches to these problems. A file can be signed or encrypted with PGP, and transported by FTP or even email. These batch processes are bulky and unsatisfying. They require action by special accounts, often initiated by a polling program run by *cron*. *Stage* provides immediate updates, initiated by the end user.

Lower level encryption, like IP/SEC and IP/V6, offer more-general solutions. We could use various existing tools if these were deployed. Unfortunately, they are not widely available yet. We needed these tools a year ago.

## 10 Limitations and further work

Although these routines are fairly straightforward, some users have prevailed with some minor enhancements. *Cput* does not have an option to run a program when the transfer is completed—a feature found in some file transport programs. In one application the receiving host must scan the receiving directory with a *cron* job looking for new files to process. These files are large (on the order of a gigabyte) and take a while to transfer. The *cron* job needs to know when the file is available. We set the file per-

missions to 0000 until the transfer is complete. The same application needed a unique file name on the destination host. An optional string ("%u") in the destination file ensures a unique file name.

These routines make no effort to deal with a file that gets shorter during the transfer. The user should ensure that the source files don't change during transfer.

*Stage* makes no attempt to lock the target directory. If two people stage to the same part of a target directory at the same time, the results are undefined. *Stage* can also overwrite programs while they are executing, causing core dumps in many versions of Unix. Since it doesn't handle special files or links, it is probably unsuitable for updating a remote root directory.

One could teach *stage* about hard and symbolic links, but it would add a lot of complexity to the program, which doesn't seem to be worth it.

There is no mechanism here for a client user to determine how much disk space is available on his external partition. The easiest solution is to install the master directory on a partition of the same size as the slave's partition. The user can monitor his inside usage. *Stage* makes no special effort to delete external files before installing new ones, so the outside partition could conceivably fill up during an update if it was nearly full.

Some users wanted more control over the update process. *Stage's* scan can take a long time, particularly if the directory tree has many gigabytes and checksumming is used. These users wrote scripts to create a list of files to update, and *xargs* can feed this list to *stage*. I had to add a parameter to suppress the descent below directories that were not mentioned on the command line, so we wouldn't do more work than these scripts wanted.

Cryptography can be no better than the quality of the keys. It is hard to generate key material with general purpose computers. I rely entirely on *truerand* to get this right. I did generate 10 megabytes of random data from *truerand* (it took several days) and had Eric Grosse, one of our local numerical analysts, run it through a suite of randomness tests. It passed.

There have been some problems reported with a slightly-restricted version of MD5 lately. Perhaps MD5 will fall soon. It is still possible that HMAC using MD5 would still be safe: HMAC frustrates some attacks on its hash primitive. In any case, I will switch to SHA1.

In general, we like these routines the way they are, and are resistant to creeping featurism. We like their simplicity, and their interaction with standard

Unix tools.

## 10.1 The Joint Ventures Problem

Joint ventures often occur between two companies that don't otherwise trust each other. Many such joint ventures only need to share a directory tree. This can reside on a neutral host somewhere. The contents of the directory tree, or the existence venture itself, may be highly proprietary.

The *stage* command offers most of the functionality needed to implement such an arrangement. *Unstage* reverses the file transfer: the remote directory is the master and the local directory is the slave. User's can share their work with these routines.

These two tools lack only a locking mechanism, which would reserve a subdirectory or file. For example, assume that two authors work for separate companies, but need shared access to the source for their book. One could lock a chapter that he is working on, and the other would have only read access to that chapter. The chapter could be staged back and the lock released.

I've tried to come up with some simple mechanism to enforce locking using file system permissions in the master directory, without a satisfactory result. It would be nice to change the owner of a locked directory, but that requires more privilege than I am willing to give the server software.

## 11 Availability

The early DES versions of these routines are freely available to AT&T and Lucent employees, and may be found on the companies' Intranets [1]. I will not attempt to distribute these.

Marcus has published his original *get* and *put* routines, with the original crypto API but not the DES routines [20].

I expect to have publication clearance for the authentication-only versions for non-commercial use in time for this conference. I am keen to release these routines to the general public. A general release will expose them to public review and possible improvement. Good cryptography and secure programming are hard to do—it is in our corporate interest to run these routines through the wringer.

See [19] for obtaining this software.

## 12 Acknowledgements

Marcus Ranum wrote the initial versions of *cget* and *cput*. Andrew Hume entrusted many gigabytes of sensitive data to early versions of *cput*, and made

several helpful suggestions. Hal Purdy ported *staged* to Windows 95. Lorette Archer, Steve Bellovin, Matt Blaze, John Linderman, Adam Moskowitz, Joann Ordille, and Bob Waldstein gave helpful suggestions and feedback on the software or this paper.

## References

- [1] <http://netlib.bell-labs.com/1127/ropes/crio.tar.Z>
- [2] <ftp://src.doc.ic.ac.uk/packages/mirror/>
- [3] Bellare, M., Canetti, R., and Krawczyk, H., *Keyed Hash Functions for Message Authentication*, Advances in Cryptology - CRYPTO 96 Proceedings, Lecture Notes in Computer Science, Springer-Verlag Vol. 1109, N. Kobitz, ed, 1996, pps. 1-15.
- [4] Bellovin, Steven M., *Using the Domain Name System for System Break-ins*, Fifth USENIX Security Conference Proceedings, pps. 199-208, June 1995.
- [5] Computer Emergency Response Team (CERT). See [ftp://ftp.cert.org/pub/cert\\_advisories](ftp://ftp.cert.org/pub/cert_advisories).
- [6] Computer Emergency Response Team (CERT), "/usr/ucb/rdist Vulnerability", CA-91:20, Oct. 1991. (superseded)
- [7] Computer Emergency Response Team (CERT), "SunOS /usr/ucb/rdist Vulnerability", CA-94:04, Mar. 1994. (superseded)
- [8] Computer Emergency Response Team (CERT), "Vulnerability in rdist", CA-96.14, Aug. 1996.
- [9] Cooper, Michael A., *Overhauling Rdist for the '90s*, Proceedings of the Sixth Systems Administration Conference (LISA VI), pps. 175-188, Long Beach, CA, October, 1992.
- [10] Joncheray, Laurent, *A Simple Active Attack Against TCP*, Proceedings of the Fifth Unix Security Symposium, pps 7-19, Salt Lake City, Utah, June 1995.
- [11] Lacy, J.B., Mitchell, D.P., Schell, W.M. *CryptoLib: Cryptography in Software*, UNIX Security Symposium IV Proceedings, USENIX Association, 1993, pps. 1-17.
- [12] Morris, Robert, *A Weakness in the 4.2BSD Unix TCP/IP software*, Computing Science Technical Report 117, AT&T Bell Laboratories, Murray Hill, NJ, February 1985.
- [13] Sellens, John, *filetsf: A File Transfer Systems Based on lpr/lpd*, Proceedings of the Ninth Systems Administration Conference (LISA IX), pps. 195-212, Monterey, CA, September, 1995.
- [14] Shimomura, Tsutomu, and Markoff, J., *Take-down*. Hyperion, 1996.
- [15] Venema, Wietse, *TCP WRAPPER: Network Monitoring, Access Control and Booby Traps*, UNIX Security III Symposium, pps. 85-92, Baltimore, MD, September 1992.
- [16] Vixie, Paul, *DNS and BIND Security Issues*. Fifth USENIX Security Conference Proceedings, pps. 209-216, June 1995.
- [17] Ylonen, Tatu, *SSH - Secure Login Connections Over the Internet*, 6th USENIX Security Symposium, pps 37-42, San Jose, CA, July 1996.
- [18] <http://www.cs.hut.fi/ssh/crypto/>
- [19] <ftp://ftp.research.bell-labs.com/ches/crio.html>
- [20] <http://www.clark.net/pub/mjr/pubs>

## Appendix - the Stage Protocol

This is the little protocol that *stage* and *unstage* use to control *staged* and the remote directory. The commands and responses are ASCII fields separated by a single blank and terminated with a zero byte.

```
send  rm fn
rcv   OK
```

Remove the given file or directory. Everything beneath the directory is removed as well. Returns either "OK", "ENOENT" (not found), or a string describing some other error.

```
send  st fn
rcv   uid gid mode mtime size
```

Return the stat of a file or directory. The mode is octal, the other values are decimal. "ENOENT" is returned if the file doesn't exist, and other strings contain a displayable error message.

```
send  cs fn
rcv   md5 checksum
```

Return the 32-hex digit MD5 checksum, or an empty string if the file doesn't exist.

```
send  pu fn
send  user group mode mtime size
send  (size bytes)
rcv   OK
```

Push a new file *fn*. It must not already exist. *User* and *group* are alphabetic, and currently ignored. *Mode* is octal, and *mtime* and *size* are decimal. The modification and access times are set to *mtime*, if allowed. Returns "OK" or a printable error message.

```
send  md fn
send  user group mode mtime size
rcv   OK
```

Create a directory with the given *mode* and *mtime* (if possible). *User*, *group*, and *size* are ignored. Returns "OK" or a printable error message.

```
send  ls fn
rcv   /fn1/fn2/.../fnn/
```

Return a list of files in the given directory, separated by slashes and terminated with a double slash. If *fn* isn't a directory, doesn't exist, or is empty, "/" is returned.

```
send  ge fn
rcv   OK
rcv   size bytes
rcv   (size bytes)
```

Get a remote file *fn*. Returns "OK" or a printable error message. If OK, return the size of the file in bytes, and the contents of the file.

```
send  ex
rcv   OK
```

Exit.

# WebGlimpse — Combining Browsing and Searching

Udi Manber,<sup>1</sup> Mike Smith<sup>2</sup>, and Burra Gopal

Department of Computer Science  
University of Arizona  
Tucson, AZ 85721  
{udi | msmith | bgopal}@cs.arizona.edu

## ABSTRACT

The two paradigms of searching and browsing are currently almost always used separately. One can either look at the library card catalog, or browse the shelves; one can either search large WWW sites (or the whole web), or browse page by page. In this paper we describe a software tool we developed, called WebGlimpse, that combines the two paradigms. It allows the search to be limited to a *neighborhood* of the current document. WebGlimpse automatically analyzes collections of web pages and computes those neighborhoods (at indexing time). With WebGlimpse users can browse at will, using the same pages; they can also jump from each page, through a search, to “close-by” pages related to their needs. In a sense, our combined paradigm allows users to browse using hypertext links that are constructed on the fly through a neighborhood search. The design of WebGlimpse concentrated on four goals: fast search, efficient indexing (both in terms of time and space), flexible facilities for defining neighborhoods, and non-wasteful use of Internet resources. Our implementation was geared towards the World-Wide Web, but the general design is applicable to any large-scale information bases. We believe that the concept of combining browsing and searching is very powerful, and deserves much more attention. Further information about WebGlimpse, including the complete source code, documentations, demos, and examples of use, can be found at <http://glimpse.cs.arizona.edu/webglimpse/>.

---

<sup>1</sup> Supported in part by NSF grant CCR-9301129, and by the Advanced Research Projects Agency under contract number DABT63-93-C-0052.

The information contained in this paper does not necessarily reflect the position or the policy of the U.S. Government or other sponsors of this research. No official endorsement should be inferred.

<sup>2</sup> Current address: Qualcomm, Inc., San Diego, CA 92121

## 1. Introduction

Browsing and searching are the two main paradigms for finding information on line. The search paradigm has a long history; search facilities of different kinds are available in all computing environments. The browsing paradigm is newer and less ubiquitous, but it is gaining enormous (and unexpected) popularity through the World-Wide Web. Both paradigms have their limitations. Search is sometimes hard for users who do not know how to form a search query so that it is limited to relevant information. Search is also often seen by users as an intimidating "black box" whose content is hidden and whose actions are mysterious. Browsing can make the content come alive, and it is therefore more satisfying to users who get positive reinforcement as they proceed. However, browsing is time-consuming and users tend to get disoriented and lose their train of thoughts and their original goals.

These two paradigms are used separately by most systems. A site may give you a link to a search page that will allow you to search the whole site. But this search will not generally take into account where you are coming from, therefore it will not incorporate any knowledge you already gained while browsing the site.

We argue that by combining browsing and searching users will be given a much more powerful tool to find their way. We envision a system where both paradigms will be offered *all the time*. You will be able to browse freely — the usual hypertext model — and you will also be able to search from any point. The search will cover only material *related* in some way to the current document. (Of course, global search may be offered too.)

Suppose that you are looking for information on network research at a certain institution. You may first go to their home page. If you search for 'network' throughout the institution, you'll get too many unrelated hits. If you search for 'network

research' you may get too few hits, because the word research may not appear in research documents (some keywords are often missing because they are implied by context). On the other hand, if you could search only the pages of the research department, or only pages related to research, then a query for 'network' may be quite relevant. You will be getting the context for your query from the pages where you initiated it. Such search is much better than the current most common way to get that information, which is to browse back and forth through dozens or hundreds of pages. After all, the computer is the one that is supposed to do most of the work, not you.

Some simple facilities of focusing the search while browsing have been employed. We have attempted limited browsing and searching facilities in our GlimpseHTTP package [1]. Since the work on GlimpseHTTP has not been published (aside from the code), we describe here some of the features that WebGlimpse borrowed. GlimpseHTTP indexes a UNIX file system, and provides search that can be focused to any part of the directory tree. For example, if you are looking for a Computer Science citation (our most popular demo [2] of GlimpseHTTP), you can browse one of 16 different categories and perform the search only for the current category. Only one index is needed for the whole archive; the CGI program identifies the page from which the search originates, and limits the search accordingly. The search pages are created automatically by GlimpseHTTP. If the information is hierarchical, and it is organized in a corresponding hierarchical file system, GlimpseHTTP works well. But its browsing and searching capabilities do not apply to arbitrary links (or to sites with information that is not neatly organized in a tree structure). GlimpseHTTP has been very successful; it is used in more than 500 sites [3], some quite major. WebGlimpse is a natural extension of GlimpseHTTP, borrowing some of its features but taking them much further.

The idea of searching only parts of a hierarchy is now used by Yahoo [4] to search only one category at a time. This is done in a very similar fashion to GlimpseHTTP, and it suffers from the same problem. For example, if you search for “mentally ill” under the “Arts” category, you will miss a site that features a “collection of art by mentally ill people”. The reason for that miss is that this site is listed under “Arts Therapy” which is listed under Arts, but only as a symbolic link (it is located in the “Mental Health” category). Yahoo probably made a good design decision not to follow symbolic links in the search, because following several such links may result in unrelated material. But following only one symbolic link from any subcategory would probably have been better. This is the kind of feature that WebGlimpse easily provides.

Other web search servers take a different approach. Both InfoSeek [5] and Lycos [6] offer “Find Related Sites.” The relationship is computed by searching the whole database for keywords similar to the major keywords of the given document. This is a traditional information retrieval approach, and it can be quite effective, but it can also lead to unusual results. (For example, we tried this feature in Lycos and looked for related sites to “Accumulated Accordion Annotations” in the “Entertainment & Leisure: Music: Instruments: Individual Instruments” category. The second match was “Comprehensive Epidemiologic Data Resource WWW Home Page” in the “Health & Medicine: Medical Research: Libraries, Databases & Indices” category. The similarity was the word “accumulated.” The third match was in the “Space & Astronomy: Image Files & Archives” category and it apparently resulted from sharing the word “annotations.”) The idea behind WebGlimpse is not only to utilize neighborhoods, but to allow search with additional keywords on these neighborhoods. In the “accordion” example above, we may want to search pages related to accordion annotations and also related to jazz.

With WebGlimpse we would have to input only ‘jazz.’ Such a query would probably have filtered out the unrelated documents that somehow ended in those neighborhoods.

Another example of focusing search to limited domain based on current browsing is the WWW-Entrez system [7] from the National Center for Biotechnology Information, which precomputes neighborhoods within the nucleotide database and within the Medline database and presents fixed links from all pages to their neighborhoods. The determination of sequence and text neighborhoods for the millions of records in the database is computationally intensive requiring weeks of CPU time. Like the Lycos and InfoSeek examples listed above, Entrez allows one to quickly explore a neighborhood from any given document, but it does not provide search within neighborhoods. We are told that they are considering adding this feature.

The next section describes the design and implementation of WebGlimpse. Section 3 presents applications of WebGlimpse, and Section 4 ends with conclusions and further work.

## **2. WebGlimpse Design and Implementation**

### **2.1. Overall Architecture**

In a nutshell, WebGlimpse works as follows. At indexing time, it analyzes a given WWW archive (e.g., a whole site, a collection of specific documents, or a private history cache), computes neighborhoods (according to user specifications), adds search boxes to selected pages, collects remote pages when relevant, and caches those pages locally. Once indexing is done, users who browse that site can search from any of the added search boxes and limit their search to the neighborhood of that page (or search the whole archive). In a sense, WebGlimpse transforms the archive into an easier-to-navigate hypertext. As its name suggests, WebGlimpse uses our Glimpse

search engine, which was modified slightly to add a few features useful for WebGlimpse. Only one index is needed for each archive (e.g., for each site). The focus to neighborhoods and the collection of remote pages are done in an efficient way at indexing time, so search is always fast.

WebGlimpse consists of several programs which perform five main steps. The first four are performed by the server (or publisher) administrator to set up an archive, and the last one is the actual user search of an existing archive. Their main features are given below, followed by more detailed descriptions.

#### **Analysis of a given archive**

Starting with a set of *root* URLs, this stage traverses local and remote pages<sup>2</sup> reachable by a path of links of a given maximum distance from the initial set. The links contained in each page are extracted, and their corresponding pages are then followed. The end result of this stage is a full graph of the whole archive, where the edges of the graph are the HTML links. The limit on the length of the traversed path can be set differently for local and remote pages. For example, one can allow unlimited distance on the local pages, but only a distance of 2 at any remote site.

#### **Collection of remote documents**

Non-local URLs are fetched and saved in a mirror file system. This is an optional step. Sometimes, local archives can be nicely complemented with data from remote sources. For example, with WebGlimpse one can collect in an archive a list of "favorite pages," or simply one's bookmarked pages. The links from these pages, and in general their structure, are preserved. This mirror file system can serve as a "hypertext book"

collected from the web.

#### **Neighborhoods computations**

Depending on how neighborhoods are defined, this step (which in practice is interleaved with the first step) builds all the neighborhood files to help with the search. We will discuss neighborhoods in detail in Section 3.

#### **Addition of search boxes to selected documents**

Selected documents with non-empty neighborhoods are identified and modified by an addition of an HTML code which provides the search facilities. It is possible to define which pages will be selected in a flexible manner.

#### **Search**

Glimpse is used for all the search routines.

We'll start with a description of Glimpse, because it serves as the basis for the whole design.

## **2.2. Glimpse**

The search engine for WebGlimpse is *glimpse* [8] (which also serves as the default search engine in the Harvest system [9]). For completeness, we'll mention here the features of *glimpse* that are relevant to the searching/browsing problem. *Glimpse* is an indexing and search software, designed slightly different than most other indexing systems. *Glimpse*'s index consists essentially of two parts, the word file and the pointers file. The word file is simply a list of all the words in all documents, each followed by an offset into the pointers file. The pointers file contains for each word a list of pointers into the original text where that word appears. A search typically consists of two stages (some searches can do with only one): First, the word file is searched and all relevant offsets into the pointers file are found. The relevant pointers (to the source text) in the pointers file are collected. The second stage is another search, using *agrep* [10], in the corresponding places in the original text. This is similar in principle to the

<sup>2</sup> We will use the term *page* to denote an HTML document corresponding to one URL.

usual inverted indexes, except that the word file, being one relatively small file, can be searched sequentially. This allows glimpse to support very flexible queries, including approximate matching, matching to parts of words, and regular expressions. These flexible queries are implemented by running `agrep` directly on the word file. The fact that the files are searched directly allows the user to decide on-the-fly how much of the match to see. Glimpse's default is to show one line per match (as in `grep`), but it can also show one paragraph or any user-defined record. This gives context to every match.

The second advantage of this design is that the pointers file can be built in many different ways. In particular, the granularity of the pointers — the precision of where they point to — can be set arbitrarily. The pointers can be to the exact locations of the words, which is similar to regular inverted indexes, to the documents (files) containing them, to whole directories, etc. The larger the granularity the more work will need to be done in the second stage (where the source is searched directly), but the smaller the index will be. Glimpse supports three types of indexes: a tiny one (2-3% of the size of all files), a small one (7-9%), and a medium one (20-30%). In the medium-size index the pointers point to exact locations, in the small one the pointers are just file names, and in the tiny one the pointers are to blocks of files.

The pointers file has another feature. Its pointers are indirect. They are indexes to yet another file, called the *filenames* file, which contains the list of all indexed files. To support WebGlimpse better, we added an option to glimpse to store with each HTML file name its title and, if relevant, its URL. WebGlimpse obtains this information directly from the result of a glimpse search. A typical search first gets offsets into the pointer file, from there it gets the indexes to the filename file, from there it collects the file names (and possibly the titles and URLs), and then in the final stage it searches those files directly. The

performance of glimpse depends on the complexity of the queries and the size of the index. Simple queries with words as patterns and Boolean operators is optimized by using hashing into the word file. Such queries generally take less than a second. More complex searches obviously take more time, but it is worth the extra flexibility. GlimpseHTTP was compared with WAIS [11] (independently of us), with the conclusions that "system requirements, administration and maintenance and, most importantly, user functionality is dramatically better than WAIS for both searching and browsing."

Glimpse is particularly suitable for our purposes because it supports very flexible ways to limit the search to only parts of the information base. Since the search is divided cleanly into two stages, searching for the words, and then going to the appropriate files, we can filter files in the second stage. This is done through two options in the search: `-f` and `-F` (the former added specifically for WebGlimpse). The first one reads a list of file names from a given file and uses only those files. The second one uses the full power of `agrep` to filter file names by matching. We'll describe later how WebGlimpse uses these options to limit the search.

### 2.3. Scanning the Archive

A WebGlimpse archive is built with a script called *confarc*. The script first sets up the right paths to the archive, the URL of the cgi-bin programs, the title of the archive, and other administrative details. More important parameters to *confarc* are the initial URLs from which the traversal of the archive will begin, whether or not non-local URLs should be collected, and the definition of neighborhoods. The traversal of the archive is done in a straightforward way. Each HTML document is analyzed to extract all its outgoing links. All links are checked for locality (using IP numbers), and traversed using depth-first search (we may change that to breadth-first-search in later versions). WebGlimpse

provides flexible facilities, through the use of regular expressions, to define which URLs/files should be included or excluded from the archive. For example, one may decide to exclude from the archive any files in a directory called /private/, except for one file there called "told-ya.html". The syntax for exclude/include follows the one used by the Harvest system [9].

## 2.4. Collecting Remote Pages

WebGlimpse provides facilities to collect remote URLs and include them in the indexing and in the search. A neighborhood can therefore include remote URLs, and users may jump through a neighborhood search to remote sites. We believe that such a facility is very important, and is sorely missing from local search facilities. Including remote pages in the search can help users make connections that would have been much harder otherwise. If you are looking for 'network research' you may find that someone is working on, say, stochastic analysis and has a remote link to references about network research. Or if you are looking for military accidents involving computers, you may find a reference to a helicopter incident with remote links to discussions on the computers aboard. This feature makes the search a global hypertext search.

The remote URLs that WebGlimpse collects are saved in separate files with a mapping mechanism from URLs to file names so the indexing and search can be done transparently. The original content of these URLs is not discarded. If space is a problem, however, then these mirror files can be removed, and the search will still be possible, because they have been incorporated into the (small) index. Glimpse has been modified to take this into account and provide a limited search (e.g., without showing the matching lines, which are not there anymore) when the original content is not available; it will still show the original URL (which is actually the typical way to show results on the web). We originally did not allow a

recursive collection of remote URLs, so that WebGlimpse would not be used as a robot. However, based on user feedback, and the fact that a multitude of robots already exist and are very easy to deploy, we now allow arbitrary remote collection. WebGlimpse abides by the robot exclusion rules.

## 2.5. Neighborhoods

When WebGlimpse traverses an archive, it also computes neighborhoods. The current version supports two types of neighborhoods. The first consists of all pages within a certain distance (the default is 2) of the current page. The idea is that WWW pages are often written with links to related pages, so the neighborhood concept is implicit. The second type consists of all subdirectories (recursively), similarly to the way GlimpseHTTP operates. The implementation of neighborhood search is quite simple. The list of all file names (recall that even remote URLs are mapped to local file names) that constitute a neighborhood of a given page is kept in one file (whose name is mapped to that page).

When a search is triggered, glimpse first consults the main index, finds the list of files with relevant matches, then *intersects* that list with the neighborhood of the given page. The neighborhood list is fetched *at query time*, and the index does not depend on it in any way, which allows easy access and easy modification if needed. For example, if a certain file or directory becomes irrelevant for some reason, all one has to do is delete its name from all neighborhood files (or from some).

In the current working version of WebGlimpse we added a compression of the neighborhood files to save additional space. This compression, which is especially designed to work with the glimpse index, is computed at indexing time through an extra program. The compressed neighborhood files are used directly with the glimpse search, so their decompression on the fly during the search is very quick. WebGlimpse

includes the compression and decompression programs, so if one wants to change any of the neighborhoods or generate them through another program, it can be easily done. We expect people to write scripts for generating neighborhoods, and this design makes such scripts completely independent of WebGlimpse itself. They can be run as post-processing steps. The neighborhood lists can further be compressed (and decompressed on the fly) to save space if that becomes a concern.

We are working on other types of neighborhoods as well as on general tools to allow people more flexibility to define their own types. One idea is to allow several neighborhoods for one page so that users can decide which one to use at query time. There will still be one neighborhood file per page, plus sets of ranges into it (e.g., smallest neighborhood contains files 1-23, second one contains files 1-39). Using classification or clustering tools, one will also be able to define a neighborhood as "all pages in this site that are in the same area."

## 2.6. Search Boxes

To further integrate browsing and searching, WebGlimpse automatically adds small search "boxes" to selected pages. An example of such a search box is shown in Figure 1. The HTML code for including the box is given in a template file; it can be easily changed to fit the preferences of the archive maintainer. The boxes are added at indexing time by adding HTML code (essentially a FORM field) to the original pages. The boxes are added at the bottom of pages, and special markers (implemented as HTML comments) are also added so that the boxes can be easily removed (WebGlimpse includes a program to do that). It is possible to customize where the boxes are added by simply moving these markers. The default is to add the boxes to all HTML pages with non-empty neighborhoods. The same kind of exclude/include facilities are available for selecting these pages. (Obviously, if a page is excluded from indexing no

search box is added to it, but the opposite may not be true in some cases.) The typical approach on the web is to provide a link to a separate search page. We decided to add boxes to all pages so that users do not need to go anywhere else for search. This minimizes the "context switching" and keeps users focused. They can see the content of the page while they compose the query.


Each box also includes a dynamic link to advanced search options. It's dynamic in the sense that it is generated on the fly and can also display the neighborhood (also generated on the fly). The search options interface is shown in Figure 2. The advanced options include old glimpse options like "case sensitivity," "partial match," and "number of spelling errors," as well as an option to jump directly to the line of the match (more on that in the next section). A very nice new option allows users to search only files updated within the last X days. The downside of adding boxes is that pages are modified, which some users may object (especially in a shared environment), and if pages are printed there is a little extra to print. We believe that in most cases this is still worthwhile. (Of course, it is easy to modify the box to make it just one link to the advanced search.)

## 2.7. The Output

The output of a query is a set of records, one for each matching file. WebGlimpse formats the results in four ways:

### Context for each match


WebGlimpse outputs the title of each matching URL with a link to it (if the file name corresponds to an HTML document), and, since glimpse provides the matching lines or records and not only the file names, all the matching lines or records are output too. In general, the records provide quick-and-dirty context for matches.






Search: ☒ The neighborhood of this page ☐ The full archive

Figure 1: WebGlimpse's search box



## WebGlimpse Search

Search: ☒ The neighborhood of 42nd Legislature - 2nd Regular Session Bill Grouping Link Page ☐ The full archive: WebGlimpse Search

String to search for:

☐ Case sensitive
 ☐ Partial match
 ☐ Jump to line
  misspellings allowed

Return only files modified within the last  days.

Maximum number of files returned:

Maximum number of matches per file returned:

Glimpse and WebGlimpse. Copyright © 1996, University of Arizona

**List the neighborhood of "42nd Legislature - 2nd Regular Session Bill Grouping Link Page"**

Figure 2: WebGlimpse's search options

**SB1375 - 422R - I Ver - Title: movie studios; sales tax incentives . ( Oct 13 1996)**

- TO TRANSACTION PRIVILEGE AND USE TAX EXEMPTIONS.
- commerce commission license for use in interstate commerce.
- (g) A person holding a privilege tax license to engage or continue in

Figure 3: An example of WebGlimpse's output (searching for "privilege and license")

### Providing line numbers

Glimpse can compute the right line number for each match, and WebGlimpse has an option (borrowed again from GlimpseHTTP) to bring the documents automatically to that line number. This is done by modifying the HTML document on the fly to insert the corresponding anchor, which is not trivial because some links may need to be recomputed as well.

### Highlighting keywords

All the matched keywords are highlighted (formatting them to bold in HTML), both in the output records and, in case line numbers are used, in the files themselves if the links are followed.

### Showing dates of modification

Starting at version 3.5, glimpse can provide dates for each file, filter by dates, and show them. WebGlimpse uses these features.

An example of an output (one match of a search for “privilege AND license” from our demo of the Arizona legislature pages) is given in Figure 3. (The date refers to our copy, not the original document.)

## 2.8. Experiments and Experience

Our experience with WebGlimpse is still quite limited. We give here some conservative numbers that we obtained, using a very complex archive. All experiments were run on a DEC Alpha 233 MHz with 64MB of RAM.

The archive for this experiment was the pages of the Arizona Legislative Information System, which includes information about budget, committees, statutes, the constitution, floor calendars, agendas, and (the most space consuming) full text of bills. This archive occupies 152.5 MB and about 20,000 files. The number of links was quite high, and we selected the neighborhoods to be within 3 hops of the given

page. (In hindsight, this is too much. The neighborhood files became too big and search was not sufficiently restricted; the indexing stage was also too slow as a result. But it gives a worst-case scenario.) We selected this archive for its complexity and the multitude of links.

The complete indexing process took 3 hours and 5 minutes. It took about an hour to traverse the whole archive and compute all the neighborhoods, an hour to analyze all HTML files and add the search boxes (we added search boxes to all files that had non-empty neighborhoods — about 85% of all files), 22 minutes to index the whole thing, and 41 minutes to compress all the neighborhood files. (We should mention that the glimpse indexing and compression is done with C, and everything else is done with Perl.) We believe that this is indeed close to the worst case, because the neighborhood structure was so complex (some neighborhoods, especially ones from the floor calendar files, contained thousands of links). The archive after indexing occupied 205.8 MB, a 34% increase, which was divided about equally between the index, the (compressed) neighborhood files, and the added search boxes.

Query times depend heavily on the type of queries. We run a few experiments with typical queries. As expected, we found no significant difference between a whole archive search and neighborhood searches. The running times, measured from the time a user clicks on the search box to the time the results page appears on the browser, range from 2 seconds for a query that matches less than 10 hits, to 12 seconds for a complex query with many hits. The corresponding times for pure glimpse search are less than half that; the rest of the time is taken to compose the HTML result page.

These numbers are taken from an early version of WebGlimpse. We put most of our efforts into making WebGlimpse work in a flexible manner. We expect the performance numbers to improve as we tune the code. More information

will be posted to the WebGlimpse home pages (<http://glimpse.cs.arizona.edu/webglimpse/>).

## 2.9. Design Decisions

We discuss here briefly the rationale behind the major design decisions we made. As we said in the abstract, our main four goals were fast search, efficient indexing, flexible facilities for defining neighborhoods, and non-wasteful use of Internet resources.

Our first design decision was to have fixed neighborhoods, computed at indexing time. There has been a lot of discussion lately of search *agents* that will traverse the web looking for specific information. While the ability to explore in real time is very attractive, we believe that at the moment the web is too slow and the bandwidth is too narrow to support wide deployment of a large number of such wandering agents. WebGlimpse can be thought of as an agent for servers, but users still search fixed indexes residing in one place. This makes the search much faster, but it limits its flexibility. In particular, neighborhoods in WebGlimpse cannot be defined by the users. It is possible, however, for the server maintainer to provide several different neighborhood definitions and to let the users choose between them. We have not yet implemented such an option.

The ability to define any neighborhood one wants is important. We put emphasis in the design to unlink the neighborhood definition and use from the rest of the system as much as possible. The neighborhoods files are consulted only at the end of the search process, and they are not integrated into the index in any way.

Fast search always conflicts with efficient indexing. The more you spend on indexing, especially more space, the faster the search. WebGlimpse was designed for small to medium-sized archives, and, like glimpse, it puts indexing efficiency slightly above search speed. Nevertheless, the search is quite fast, and indexing

can sometimes take quite a long time. Indexing can be slowed down by two features: 1) having to fetch many remote documents (nothing we can do about that), and 2) having to compute and manipulate complex large neighborhoods.

User interface is very important to any search application. We believe that our design of the output of queries — with the inclusion of matched lines, highlighted keywords, and dates of modification — will be very helpful. Being able to quickly judge the relevance of the output to one's query is a problem in many search services today. One often finds oneself spending considerable time following up on the multitude of links that are returned as results of queries.

We believe that making the search box available all the time, rather than the more typical link to a different search page, is a good idea. A search box does not take much “real-estate,” and it allows users to compose their search query *while* looking at the page. We received comments from web administrators who do not want to modify existing pages to add the search boxes. This is a valid concern, especially in sites where pages are owned by many people. But many sites strive for coherent design and adding boxes is not much different than requiring a certain format or adding uniform links to the main search page. We provide easy tools to add and remove those search boxes at any time.

## 3. Applications

The main application of WebGlimpse is, of course, to provide search for collections of hypertext files. We foresee several other related applications.

### 3.1. Building Personal or Topical Collections

In a sense, WebGlimpse is a “light” version of Harvest [9]. It does not have the full power of Harvest to automatically collect massive information from given sites and extract the

indexed text from different formats. But it does allow one to collect and organize specific documents that are relevant. In contrast with Harvest, the link structure of that information is kept. (Harvest, like most global Internet search facilities moves everything it collects into a flat structure.) Hypertext “books” can be written much more easily with WebGlimpse, which will collect and index all links (citations) and allow flexible browsing through the whole book. Lists of “interesting sites” are commonly kept by many people, some are quite substantial. WebGlimpse will allow the maintainers of such lists to easily make them searchable with full-text search.

### 3.2. History and Cache Files

Most web browsers cache the pages they fetch. Like typical caches, this is done completely transparent to the user, and is done for performance purposes only. But having this large set of mostly relevant pages can be very helpful. For example, two years ago we designed a system called *Warmlist* [12] that cached pages per user demand, indexed them, organized them, and provided full text search. It was meant to be a natural extension of the “hot list” concept. There are now similar commercial products, which also allow to cache everything automatically and to view and navigate based on this cache. This becomes a history feature more than just a cache. With WebGlimpse, you can easily construct an *archive* of your history list. Not only can you browse and search it, but you may also discover relationships between pages — by viewing the neighborhoods — or other context information.

### 3.3. Visualization and Customization

Combining WebGlimpse with graph drawing packages (such as [13] and [14]) will allow for better visualization of the hypertext structure. Imagine adding to the results of queries some

summaries of the documents, icons of them, or other useful information of the kind you find in static pages with links to related documents. When you visit a page and perform a query (and queries with WebGlimpse can be simpler because the context is kept and the domain of the search is smaller) you get a customized view of the “way ahead”. This view may be just as good in terms of information as the static view, but much more relevant to you. In other words, you as the navigator can build your own hypertext part of your way, customized to the areas of interest to you. In particular, it would be very interesting to see how to combine our fixed neighborhood search with the ideas of scatter/gather [15], or an automatic classification system like AIM [16].

## 4. Conclusions

Searching the web is growing quickly from an infancy stage. The current facilities are quite amazing and very useful, but they are far from being the last word or even a good word. Finding useful information on the web is still a very frustrating process. We believe that more methods need to be attempted, more prototypes employed and experimented with, and more paradigms need to be explored. WebGlimpse presents one such attempt. It is simple, easy to build, natural to follow, and flexible so it can be extended. WebGlimpse is part of the FUSE (Find and USE) project at the University of Arizona (<http://www.cs.arizona.edu/fuse/>), where we are working on other methods for the same problems.

## References

- [1] The Home Page for GlimpseHTTP, <http://glimpse.cs.arizona.edu/ghttp/>
- [2] Glimpse Search of Computer Science Bibliography, <http://glimpse.cs.arizona.edu/bib/>
- [3] A partial list of sites using GlimpseHTTP, <http://glimpse.cs.arizona.edu/ghttp/sites.html>
- [4] Yahoo search, <http://www.yahoo.com/>
- [5] InfoSeek Search, <http://www2.infoseek.com/>
- [6] Lycos Internet Directory, <http://a2z.lycos.com/>
- [7] Epstein J. A., J. A. Kans, and G. D. Schuler, "WWW Entrez: A Hypertext Retrieval Tool for Molecular Biology," *Proceedings of the Second International World Wide Web Conference*, Chicago, Illinois (October 1994).
- [8] Manber U. and S. Wu, "GLIMPSE: A Tool to Search Through Entire File Systems," *Usenix Winter 1994 Technical Conference*, San Francisco (January 1994), pp. 23–32. See also Glimpse Home Pages at <http://glimpse.cs.arizona.edu/>
- [9] Bowman C. M., P. B. Danzig, D. R. Hardy, U. Manber, and M. F. Schwartz, "The Harvest Information Discovery and Access System," *Computer Networks and ISDN Systems*. **28** (1995), pp. 119–125.
- [10] Wu S., and U. Manber, "Fast Text Searching Allowing Errors," *Communications of the ACM* **35** (October 1992), pp. 83–91.
- [11] Morton D., and S. Silcot, "Systems for providing searchable access to collections of HTML documents," *First Australian WWW conference*, July 1995. (<http://www.scu.edu.au/ausweb95/papers/indexing/morton/>)
- [12] Klark P., and U. Manber, "Developing a Personal Internet Assistant," *Proceedings of ED-Media 95, World Conf. on Multimedia and Hypermedia*, Graz, Austria (June 1995), pp. 372–377.
- [13] Ayers, E. Z. and J. T. Stasko, "Using Graphic History in Browsing the World Wide Web," Technical Report GIT-GVU-95-12, Georgia Inst. of Technology (May 1995).
- [14] Domel, P., "WebMap - A Graphical Hypertext Navigation Tool," *Proceedings of the Second International World Wide Web Conference*, Chicago, Illinois (October 1994).
- [15] Cutting D. R., D. R. Karger and J. O. Pedersen, "Constant Interaction-Time Scatter/Gather Browsing of Very Large Document Collections," *Proceedings of the Sixteenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (1993), pp. 126–134.
- [16] Barrett R. and T. Selker, "AIM: A New Approach for Meeting Information Needs," IBM Technical Report RJ 9983 (October 1995).

# Mailing List Archive Tools

Sam Leffler, *Silicon Graphics, Inc.*  
Melange Tortuba, *Tortuba Consulting*

## Abstract

Electronic mailing lists are a common forum for discussion on the Internet. Each participant receives a copy of each message posted to the list with an additional copy typically archived at a well-known site for later retrieval. Busy lists and long-running lists can quickly accumulate many messages making it difficult to locate information. Few sites provide mechanisms for the efficient storage or retrieval of information from these archives. The Mailing List Archive (MLA) tools were designed to address these issues. Mail messages are stored in a space-efficient manner and the archive information is processed to create a database that is optimized for fast queries. The MLA tools are designed for use as Computer Gateway Interface (CGI) applications so that archives can be accessed through HTML forms on the World Wide Web. Concurrent update and retrieval are an integral part of the design so that up to date information is always returned in query results.

## 1.0 Introduction

UNIX mailing list archives are typically unstructured files that hold the concatenated mail messages stored in the order in which they were received. This simple format makes it easy to apply text-oriented UNIX tools such as `grep` and `awk` but fails to capture much of the structure of the discussions that transpire on a list. Furthermore, when mailing lists are especially active archives can grow very large, causing text-oriented tools to slow down noticeably. Bulletin board systems such as Usenet together with threaded reader programs attempt to address these problems but require nontrivial administration and the use of involved packages for transmitting and storing messages. For many discussion lists electronic mail is the preferred medium, and what is needed is a simple to

use backend system for archiving messages. While it is possible to feed a mailing list into the normal Usenet software for the purposes of creating an archive, the MLA tools were created as a much simpler solution that is easier to maintain and more efficient.

The MLA tools consist of a program that stores mail messages in a mailing list archive database and several programs for doing queries from a database. Additional tools are provided for doing administrative tasks such as removing messages that are old (i.e. expired) and for optimally compressing a database (most compression is, however done on-line). User access to an archive is possible through command-line programs, electronic mail, or HTML forms on the World Wide Web. Message relationships (i.e. threads of discussion) are maintained and query and navigation tools preserve this threading structure even when navigation is done with an HTML browser.

The MLA tools differ from other packages in several important ways:

- They work with relatively unstructured mail messages (as opposed to news postings, which have structured information that simplifies cross-referencing messages). Programs such as hypermail [1] also work directly with mail messages but do not do as good a job of deducing message relationships, do not work on-line, and do not provide a query interface.
- They are intended to be simple to use in unprivileged environments. The tools require no special-purpose protocols or system registration of services.
- They are designed to efficiently support database updates without impacting concurrent queries. The MLA tools are designed with the express intent of doing on-line updates for each mail message.

- They are simple to use within the framework of the World Wide Web by way of a template-based facility for generating HTML documents.

The MLA tools might best be contrasted with hybrid systems built from an indexing engine and a CGI interface to a threaded news server. A system such as *ni* [4] has similar functionality but lacks the navigational support provided by the MLA tools.

The next sections of this paper present the tools and provide examples of how they are used. Subsequent sections discuss the design of the database used to store archives, issues in parsing and building threaded mailing list archives, and issues in building a useful navigation interface for the World Wide Web. These are followed by a section reporting performance results and, finally, a summary and discussion of future work.

## 1.1 Creating a Mailing List Archive

The *mldapdate* tool is used to create and update (add messages to) an archive. Typically, an MLA is created from an existing archive of mail messages and then a feed is setup so that new traffic is directly entered into the database. Old message archives are assumed to come in three forms: mbox-style files in which mail messages are concatenated and the UNIX-style "From" message lines are still present, MH folders where each mail message is written to a separate file and tagged with a "Delivery-Date" header line, and Usenet news articles where a "Newsgroups" line is treated as a "To" address. To convert an mbox-style file to an MLA, a command of the form:

```
mldapdate foo-archive
```

might be used. This creates an MLA in the current working directory, if it does not already exist, and enters the mail messages found in the file *foo-archive*. *mldapdate* can also read from standard input; for example,

```
zcat foo-archive.Z | mldapdate -
```

can be used to supply a compressed archive to *mldapdate*.

Alternatively, the following enters all the messages from an MH folder:

```
mldapdate -y ~/Mail/inbox/*
```

(The *-y* option to *mldapdate* forces it to not scan the input text for an initial message separator pattern, typically a UNIX-style "From line" as generated by the normal system mail delivery program. In this case the *-y* option is needed because MH removes UNIX-style "From lines" when mail messages are incorporated into folders.)

Messages are stored in a compacted form with the useful header information written to a table of contents database file that is optimized for searching and the body written to a message-body database file. Message bodies that exceed a certain threshold size are written to separate spillover files instead of the message body database to optimize space usage in the hashed message body database. All message bodies, whether they appear in the database file or in a spillover file, are stored in a compressed form using a PKZIP-style compression algorithm [8].

In addition to the message header information, the table of contents database also includes inter-message relationships such as whether a message is a reply to another message. This information is used to build thread relationships between messages: all messages that appear to be part of a single thread of conversation are identified and query results may provide messages organized as threads, as opposed to unrelated collections of messages.

## 1.2 Feeding an Archive

A key component of the MLA design is the ability to directly connect a live feed to the database. That is, mailing list traffic can be immediately entered into an archive with the thread-related information automatically updated and available for queries. The typical way to effect this is to setup a mail alias that invokes the *mldapdate* program to enter each mail message in an archive. Alternatively, mail messages can be fed to *mldapdate* through personal delivery tools such as the *slocal* [3] or *procmail* [5] programs. Each linkage mechanism has advantages and disadvantages; most importantly, delivery via a mail alias may require that the MLA files and containing directory be owned by the user that invokes *mldapdate*. Invoking *mldapdate* through personal delivery tools permits file protections to be setup in a more private manner. The send-mail delivery route can be overcome by using a setuid wrapper program that invokes *mldapdate*. Note also that ownerships and protections must permit read access to the CGI applications invoked by the HTTP server.

The following *.maildelivery* entry might be used with the MH *slocal* program to feed incoming messages for a "flexfax" mailing list directly into an archive:

```
sender owner-flexfax | R \  
"mldapdate -y -d arch/fax -"
```

(the *-y* option is supplied in case *slocal* removes the "From" line; see above). An equivalent mail alias would be:

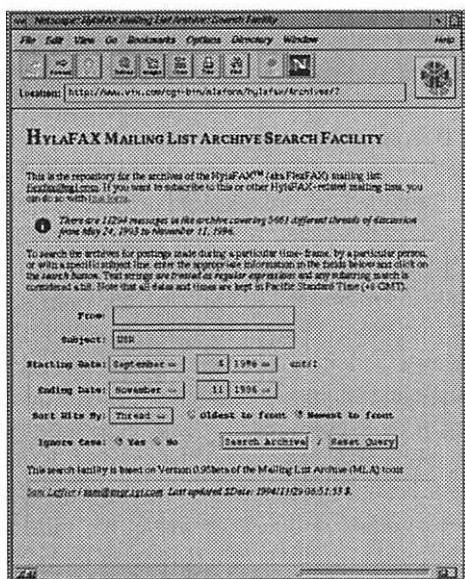


Figure 1. Form generated by mlaform.

```
flexfax-archive: "/bin/mlaupdate \
-d /fax/Archives -"
```

Note that in this situation the -y option is not needed because the "From line" is known to be present in the message delivered to mlaupdate.

If a mailing list has a very high traffic rate and archive updates cause excessive locking of the database, then the mlabatch program can be used in place of mlaupdate. mlabatch works by holding onto mail messages for a period of time before entering them in a database. If multiple messages are received before an update is scheduled, then they are batched together and only a single update of the database is done (thereby reducing the time the archive is kept locked). mlabatch supports all the same command line options that mlaupdate does as well as two additional options for controlling the batching operation. Thus an archive feed can be converted to use batching simply by invoking mlabatch instead of mlaupdate.

### 1.3 Querying an Archive Through the World Wide Web

Three tools are provided for querying an archive through the World Wide Web: mlaform, mlaquery, and mlafetch. These programs comply with the Computer Gateway Interface (CGI) protocol [12] and so may be invoked directly from other HTML pages. mlaform is used to generate an HTML page from which a query is constructed. mlaquery implements database queries returning a specially formulated HTML page that includes a hidden form of the query result in each HTML link. mlafetch retrieves a mail

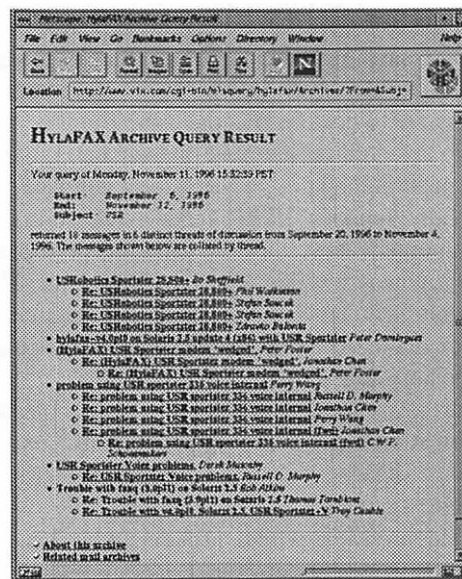
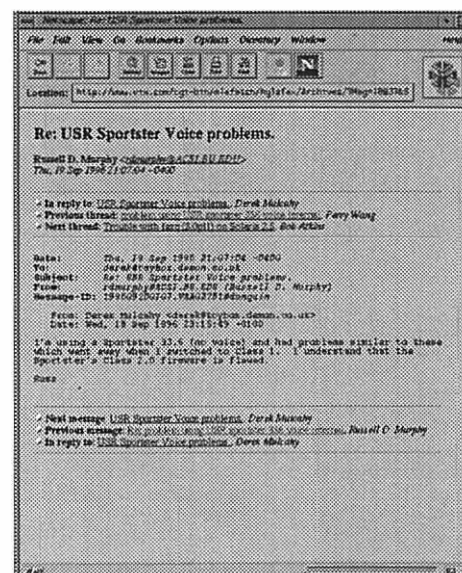


Figure 2. Query result generated by mlaquery.

message from an archive, also encoding query results in links for navigational purposes. In normal use, a user is presented a query form generated by mlaform (see Figure 1), constructs a query that is performed by mlaquery (Figure 2), and then retrieves articles using mlafetch (Figure 3). As such the design of these three programs assumes that they will be used together.

mlaform is typically used as the top-level interface to an archive. It generates an HTML page based on a template file and the underlying MLA database. Template files are HTML documents that contain escape codes that are replaced by an MLA

Figure 3. Article retrieved by mlafetch.



tool. Interesting information such as the number of messages in the archive and the ages of messages in the archive are available through escape codes. Escape codes are also provided for formulating queries that are to be done by mlaquery. The fragment of the flexfax archive template shown in Figure 4 gives the general flavor of an MLA form template.

mlaquery supports HTML query-style forms; it takes its arguments from the command line or through shell environment variables as specified by the CGI specification. Searches are done over a single archive with queries constructed from phrases that constrain the returned messages:

- match a regular expression or boolean combination of expressions against mail message header lines (e.g., messages from "sam@.sgi.com"),
- consider messages within a specific time frame, and
- ignore upper-lower case text distinction in matching strings.

The matching messages are returned collated according to thread, author, subject, or date. The maximum number of hits returned can be controlled as well as the maximum depth of the threading information returned when query results are collated by threads.

The HTML document generated by mlaquery includes an encoded form of the *query hit set*—the set of messages that satisfied the query. This information is critical to the navigational support provided by the tools and is described more in Section 2.3.

mlafetch returns a formatted message together with links to other messages in the query hit set and to other messages in the database that are related to the message; e.g., if a message is a reply to another message then the original message can be viewed even if it was not matched by the query. mlaform propagates the encoded information passed to it by mlaquery so that the context of the original query is always present. Like mlaform and mlaquery, the format and content of the HTML is defined by a template file that administrators can tailor to their needs.

**Figure 4. Sample MLA form template.**

```
<PRE>
<B>          From:</B>%*F
<B>          Subject:</B>%*S
<B>Starting Date:</B>%*+< <I>until</I>
<B>  Ending Date:</B>%*+
<B>  Sort Hits By:</B>%*+C <INPUT TYPE=submit
VALUE="Search Archives">/ <INPUT TYPE=reset
VALUE="Reset Query">
</PRE>
```

## 1.4 Administering an Archive

Archives are for the most part self-maintaining. Data structures are designed to grow efficiently as new messages are added to the database. There are, however some components of the database that can be compacted if the database is manipulated off-line and tools are provided to do this. Otherwise, there are tools to delete messages from an archive and low-level tools for interrogating the internal data structures used in an archive—mainly for the purpose of debugging problems. Each MLA is stored in a separate directory and may be copied with the normal UNIX command line tools. However using programs such as cp or tar to copy an MLA may generate a copy that uses more disk space than the original because the hashed database package used by the MLA tools utilizes holey files. If this is an issue, the mldump tool can be used to emit an ASCII form of the archive that can then be fed back into mldump. Thus an archive can be copied by commands of the form:

```
mldump -d foo.orig first-last | \
mldupdate -d foo.copy -
```

## 2.0 MLA Design and Implementation

The MLA tools were designed to be efficient both for queries and for updates, but the expectation was that far more queries would be done than updates. It was very important however, that on-line updates be supported so that queries always return current information; otherwise, the tools would not be used in place of existing facilities.

An archive consists of at least two files: the table of contents database, and the message body database. Message bodies that are too large to fit directly in the message body database file are written to spillover files, with the entry for the message setup to reference this file. The table of contents file is the critical data structure in obtaining high performance for both queries and updates; it is managed with a special-purpose package. The message body database is implemented using a publicly-available hashed database package [9]. Message bodies are compressed with a PKZIP-style compression algorithm whether they are stored directly in the database file or in spillover files; this is done with the publicly available zlib package [10].

### 2.1 Table of Contents Database

The table of contents database (TOC) holds all the information about an MLA except for the mail message bodies. The data structures are designed so

that programs that do queries can map the file contents into memory in a read-only fashion and do a lookup with minimal overhead. Specifically, the number of pages that must be touched is typically small and no parallel data structures or excessive calculations are required of the application as a result of the memory-mapped style of accessing the data. TOC updates also use a memory-mapped interface to the data and are designed to be done in-place for the vast majority of updates (when certain data structures overflow the database is grown to accommodate more information and this operation can be expensive to carry out). The efficiency of normal queries and updates is important in minimizing TOC lock contention when queries collide with updates.

Each TOC has a fixed size header followed by several tables and a *string pool*. The string pool is a data structure that holds all the ASCII strings retained from the headers of mail messages stored in the archive. Redundant strings and substrings are shared so that, for example, subject lines are trivially compressed. The other tables in the TOC hold the following information:

- a message descriptor table that has one entry for each message in the archive,
- a thread table that identifies the top-most message in each thread of discussion,
- a sorted message table that references entries in the message descriptor table sorted by message delivery time, and
- a reply spillover table that holds message reply information that does not fit in the normal message descriptor data structure.

Tables are initially sized according to a set of heuristics and configurable parameters; these parameters are stored in the TOC. From that point onward the tables are automatically grown whenever entering a new message would fill up a table. The revised table sizes are calculated to reflect the existing usage patterns, with each table guaranteed to be no more than 75% full. The initial table sizes are based on the following assumptions:

- the number of threads of discussion is at most 1/3rd the number of messages, i.e., on average there will be no more than two replies for any message;
- the size of the reply spillover table is 1/6th the number of messages, the expectation being that most messages get a single reply that can be stored directly in the message descriptor;
- the string pool gets an average size message (administrator defined) for 1/2th the maximum number of messages (since most messages are replies there is a significant sharing of strings

for the subject, user names, mail addresses, etc.).

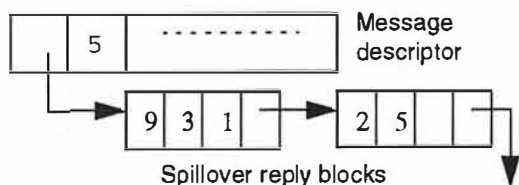
By default, mlaupdate creates new archives with space for 2500 mail messages, and assumes that an average message requires 128 bytes of space in the string pool. These numbers were chosen empirically. At an average of 100 mail messages per day, these initial sizes accommodate a mailing list for 25 days before the database must be expanded. After that, the database will need to grow about once a week if traffic patterns remains consistent.

Each mail message in an MLA is assigned a unique number that is used for all references (rather than file offsets that would need to be updated when data structures grow). Likewise each distinct thread of discussion is given a unique number that is used for all references. String references are stored as offsets into the string pool and must be updated if a string is moved as a result of the string pool being compacted.

The two important pieces of information that need to be tracked are the mail message contents (headers and body) and the message relationships that define the discussion threads. Each message descriptor contains references to the strings that make up the headers; the message body is looked up separately using the unique message number assigned when the message is added to the database. Message descriptors include references to the message's parent message (if it exists), top-level thread (as stored in the global thread table), and a list of messages that are considered replies to this message.

Message replies are handled specially. Each message may have a single reply that is stored directly in the message's descriptor. However, if multiple replies exist then the reply information in the message descriptor points to a list of reply blocks that are stored in the spillover reply table. Reply blocks are always allocated in the reply spillover table in groups of four. If all the replies for a message can be stored in the available space then they are; however if there are more replies than will fit, three replies are recorded and the fourth entry in the block is used to reference another block in the table where the remaining replies are stored. Figure 5 shows an example where a message has five replies.

The scheme used to manage replies provides an efficient mechanism for handling messages with a small number of replies, but can incur noticeable overhead when a single message has many replies since 25% of each reply block in the spillover table is spent on linking to the next block. Various schemes and reply block sizes were considered before choosing this design based on statistics collected from a variety of mail archives.



**Figure 5. Message reply data structures.** Message 31 has 5 replies so the in-descriptor reply points to a block in the reply spillover table. Two 4-element blocks in the spillover table are allocated but only five of the six possible entries are actually filled in/valid.

## 2.2 Message Parsing and Thread Construction

The MLA tools expect mail messages that conform to [6]. When a message is to be added to an archive, the header information is extracted and the body is stored in the message body database. Only the interesting header lines in a mail message are retained; all others are discarded. Mail message bodies can be stored preformatted as HTML or in their original form. Storing messages as HTML is most efficient if queries are to be done through the World Wide Web, as the overhead of parsing the body to insert HTML escapes (e.g., "&gt;" for "<") and for recognizing and inserting URLs can be done once (rather than each time the message is retrieved for display). It is also possible to retrieve message bodies with embedded HTML directives removed, but this can remove information if the original message included HTML directives.

The majority of the effort involved in entering a message into the archive is in establishing the relationship to other messages; specifically, determining if a message is a reply to another message and, most importantly, which one. This problem is difficult because there are no standard mechanisms used to identify the target message to which a message is a reply. Reliable reply information can usually be found in the "In-reply-to" and "References" lines in a header, but not always. It is not uncommon for users to use an existing mail message as a convenient means for formulating a new posting to a mailing list (to avoid having to enter the "To" address for the message); this can cause anomalous results in formulating message relationships. Also, some mail systems do not include either type of reply information, leaving only the hint of a relationship in the "Subject" line of a message (e.g., a leading "Re:" in the subject information).

Reply information is handled by the following scheme:

- Look for an "In-reply-to" field in the header.
- If no In-reply-to field is present, use any "References" header information.
- If no In-reply-to or References information is present, consider the "Subject" line.
- Extract identification information from the reply information (In-reply-to, References, or Subject). Note that doing this can be problematic because the format of these strings is not standardized.
- Search the archive for an existing mail message with matching identification and if the parent message was identified by something other than the Subject line, compare the subject information to weed out postings that are false relationships (see above).

This scheme can still link messages into a thread that do not belong, but such messages are usually impossible to deduce without actually interpreting the meaning of the message (e.g., someone that uses one posting to create an unrelated posting, leaves the subject line the same, but the content of the posting is totally unrelated to the original).

## 2.3 CGI Support and Navigation

The MLA tools are designed to be used through the Computer Gateway Interface (CGI) protocol that is used to connect static URLs to programs that execute on a server machine. Programs intended for use in this way all work as backends to query-style HTML forms (i.e., they take a collection of arguments and return an HTML document that is the result of applying the arguments to an MLA). The format and content of the information returned by a tool is defined by a template file that contains HTML directives and, optionally, escape codes that are replaced "on the fly" according to the contents of an archive or the result of a query.

An interesting aspect of the work done in this area is how the tools provide good navigational support in accessing the messages that comprise a query hit set. The basic problem is this: Given a query, how can the results be made available to clients across the World Wide Web in such a way that the context of the query result is maintained across multiple HTML pages. This problem is difficult because the underlying HyperText Transmission Protocol [7] is stateless and so any context that is to be maintained across multiple HTTP connections must be stored in the client, possibly in reference to some state on the server. In this case the state to maintain is the result of doing a query of the archive

```

11236 11243 11245 11248 11264 11237 11238 11239 11240 11253 ... 11271 11273 11272 11278 11279 11280 11281
└─▶ 11236 +0 +7 +9 +12 +28 +1 +2 +3 +4 +17 ... +35 +37 +36 +42 +43 +44 +45
    └─▶ NHits: 47 MinMsg: 11236 BitRange: 6 MsgDeltas: 0, 7, 9, 12, 28, 1, 2, 3, 4, 17, ...
        └─▶ AAM5SmAAAC5OB0SeyJAdIOa1DZAc...XcKdBgUaq2DgSF6UAAAebv

```

**Figure 6. Encoding a hit set.** The set of message numbers is encoded as an ASCII string for embedding in URLs. First the set of numbers are converted to a base value and a set of positive offsets. Then the numbers are encoded as a bit string in which the offsets are bit-packed. Finally the bit string is converted to ASCII using a restricted alphabet.

database. This state can be stored in the client in one of several ways:

- as a query that must be redone each time the result is needed,
- as a reference to state cached on the server,
- as the result itself.

The first two options might be the same if query results are cached with the query string used as a key to reference cached results. This has the important characteristic that if the cached result on the server is no longer available then it can be recreated by redoing the query using the supplied key/query string. If query strings are large however then this scheme can be expensive. Also, caching state on the server potentially requires unlimited server resources to store results. Systems such as Alta Vista [11] and DejaNews [13] appear cache state on the server.

The last option, storing the query result in the client, works well only if the result is small enough that it does not cause a large HTML document to be generated. There is also the subtle requirement that results may be stored in a client for very long periods of time—months, maybe years—so references encoded in the result must work even if the database has changed.

The MLA tools use the third scheme to manage query results. The set of message numbers that comprise a query hit set are returned to the client in an encoded format. This is combined with some careful usage of the HTML language to minimize the size of HTML documents that are returned as a result of a query.

An encoded hit set is an array of message numbers. This information must be returned as an ASCII string comprised of a limited set of characters that are legal to pass as part of a URL. The set of hits is transformed to a 32-bit base message number and a set of offsets that are relative to the base message number. Offsets are expressed using the minimum number of bits required to represent the positive values. A fixed header is then prepended to this information and the entire result is encoded as ASCII using a restricted alphabet. This scheme works especially well when queries are constrained to reasonable time ranges since message numbers in the

hit set are then closely packed. Figure 6 shows an example of this procedure.

The encoded hit set is used by programs like mlafetch to generate context-sensitive linkage such as the next message in a query result; linkage that could not otherwise be done without redoing the original query. Note that a side-effect of storing the result in the client is that if the contents of the database change while the client is navigating an HTML page, then the navigated view will remain unchanged even if redoing the query might yield a different set of hits. This is considered good from the standpoint of a user interface designer, though some people might prefer the alternative.

## 3.0 Performance

### 3.1 Space Efficiency

To save space in the archives, the MLA tools do the following when a new message is entered:

- unneeded mail headers are removed,
- the message body is stored compressed,
- common strings in the header are shared through the string pool.

This saved space is offset by the space required to store message relationship information and general overhead in the TOC database related to efficient operation. In addition there is overhead associated with the hashed database used to store message bodies.

Usage statistics for several archives are shown in Table 1. The flexfax archive contains messages for about three years worth of postings to a fairly quiet mailing list (about 10 mail messages a day). The majordomo archive contains about four years of postings to the majordomo users list. The sgi.bad-attitude archive holds about four years of news article postings to an internal Silicon Graphics newsgroup. The www-vrml archive is about a year and a half of postings to the www-vrml mailing list.

Tests were run on a Silicon Graphics Indy workstation running IRIX 6.2 with the XFS filesystem (a filesystem that supports files with holes in them) and version 0.95 of the MLA tools. The

| Archive          | Messages |       | Threads |      | Replies |       | String Pool |        | Spillover Files | Space Usage |        |
|------------------|----------|-------|---------|------|---------|-------|-------------|--------|-----------------|-------------|--------|
|                  | Total    | Max   | Total   | Max  | Total   | Max   | Total       | Max    |                 | Data        | MLA    |
| flexfax          | 11265    | 14293 | 5645    | 7162 | 4264    | 5388  | 1.14MB      | 1.43MB | 226             | 31.6MB      | 15.5MB |
| majordomo        | 10310    | 11481 | 5158    | 5890 | 4488    | 4840  | 1.08MB      | 1.43MB | 79              | 23.7MB      | 13.9MB |
| sgi.bad-attitude | 12317    | 14524 | 1571    | 1875 | 13136   | 15440 | 0.79MB      | 1.16MB | 55              | 69.6MB      | 12.5MB |
| www-vrml         | 5470     | 5979  | 1975    | 2191 | 3896    | 4212  | 0.49MB      | 0.59MB | 126             | 14.4MB      | 9.4MB  |

**Table 1: Sample MLA statistics.** The sizes of data structures are shown as the number of items in use (**Total**) versus the number of items to be filled before the TOC database must be expanded (**Max**). Spillover files are files created to hold message bodies that exceed the maximum size of a message that is stored directly in the message body database. Space usage gives the total amount of disk space used by the uncompressed input data (**Data**) and the final MLA (the TOC database, the message body database, and the spillover files).

hashed message body database files were created with version 1.85 of the Berkeley DB software. Other than a 3 kilobyte threshold for creating spillover files the default parameters were used in creating the archives.

The implementation uses a 56 byte message descriptor and 32-bit numbers for all thread and message numbers. The fixed size header at the front of the TOC is about 100 bytes. This results in TOC files that are very small in comparison to the space used to store message bodies; c.f. Table 2.

Two specific areas of space usage in the TOC were examined: the string pool and the reply spillover table. Table 3 summarizes statistics for the sample archives related to these two areas. The string pool was generally very effective in saving space; on average nearly half the space required to store the header strings was saved by sharing strings. This reflects the fact that most postings have at least one reply and so the subject strings in the messages can be shared. The more replies there are to a posting, the more sharing that will take place. Note however that optimal sharing requires periodic off-line compaction of the string pool since reply messages chronologically follow the initial posting so the

subject string in the original posting cannot be compacted until after a reply is received.

The sample archives examined had very few unspecified header lines. As a result there were few null strings in the archive so the default string sharing for this special case was ineffective. This result would likely change if more header information were retained in the TOC database.

The overhead for the linked list scheme for storing reply information is more than 30%. That is, more than 30% of the space allocated to storing reply information is either used for links to following blocks in the spillover table or not used at all (i.e. unused entries in the last block of a list). This number indicates that most reply spillover blocks are fully populated since if every block were filled this figure would be approximately 25%. While the overhead may seem high, the actual amount of storage associated with this waste is relatively insignificant in the overall totals. An alternate scheme that allocates contiguous arrays in the spillover table might be more efficient, but would significantly increase the complexity of the software.

| Archive          | Input | TOC  | MsgBodies |       | Total |
|------------------|-------|------|-----------|-------|-------|
|                  |       |      | DB        | Spill |       |
| flexfax          | 31.6  | 2.06 | 12.0      | 1.44  | 15.5  |
| majordomo        | 23.7  | 2.00 | 11.2      | 0.70  | 13.9  |
| sgi.bad-attitude | 69.6  | 1.91 | 10.2      | 0.44  | 12.5  |
| www-vrml         | 14.4  | 0.92 | 7.6       | 0.88  | 9.4   |

**Table 2: Sample MLA space usage.** All numbers are megabytes of disk space allocated (no holes). Input refers to the raw input data. TOC is the table of contents database. Message body data is broken up into the space used by the hashed database file and separate spillover files.

| Archive         | String Comp. | Null Strings | Reply Overhead |
|-----------------|--------------|--------------|----------------|
| flexfax         | 1.88         | 13           | 32%            |
| majordomo       | 1.87         | 55           | 35%            |
| sg.bad-attitude | 2.60         | 5            | 30%            |
| www-vrml        | 1.97         | 2011         | 32%            |

**Table 3: Sample MLA space efficiency.** String Compression is the ratio of the space required to store strings without the string pool to storage with the string pool. Null Strings is the count of empty strings. Reply Overhead is the percentage of reply entries not used or used to link blocks in the spillover table.

### 3.2 Update Performance

Table 4 shows results for some sample update tests. The tests were all run on an SGI Indy system running IRIX 6.2 with the archive stored on an XFS filesystem. The machine had a 133MHZ R4600SC with 96MBytes of memory.

The flexfax, majordomo, and www-vrml archives were built from several gzip-compressed files that each contained multiple messages. The raw data for the sgi.bad-attitude archive had each message in a separate file; this is more indicative of normal usage where mlaupdate processes mail messages as they are received. In all test cases the average real time to process a message is less than a tenth of a second; this is consistent with the goal to keep update time small so that lock contention on the MLA is minimized for interactive users doing queries.

All the work that involves the table of contents database is done by directly manipulating a memory-mapped copy of the TOC file. The most expensive work is determining the thread relationship. If the message appears to be a reply then it may be necessary to search backwards chronologically over message headers to find a message's parent message. In normal operation adding a message to an archive involves touching only a few pages of the TOC file. In the worst case most of the string pool and message descriptor table must be touched. The sgi.bad-attitude test shows the most pronounced effect of this requirement; many messages in this archive are replies and the update time is noticeably higher than a similar-sized archive such as flexfax.

As noted in Section 2.1, allocating entries in the various TOC tables is typically inexpensive; just incrementing a counter in the TOC header. When the addition of a message would cause one of these tables to overflow, all the tables are expanded in-place; this operation can take several seconds depending on the characteristics of the underlying system (when there is sufficient memory to cache the contents of the TOC the grow operation can happen very quickly).

| Archive          | Total Time |        |        |
|------------------|------------|--------|--------|
|                  | User       | System | Real   |
| flexfax          | 163.86     | 51.22  | 259.75 |
| majordomo        | 131.25     | 41.02  | 196.54 |
| sgi.bad-attitude | 217.19     | 61.02  | 423.52 |
| www-vrml         | 82.02      | 23.62  | 135.88 |

Table 4: Build times for sample MLAs. All times are in seconds.

One thing to note about the operation of the mlaupdate program is that it is faster to process multiple messages together than to add each individually. This is because mlaupdate builds hash tables for message header strings to use in determining message relationships. When mlaupdate is given one message at a time to add to an archive it is likely these tables will be built multiple times (whether or not there is overlap depends on whether the messages are related).

### 3.3 Query Performance

MLA queries are done as a result of URL lookups that cause the MLA tools to be invoked as CGI applications. Consequently user perception of query performance is dominated by the overhead associated with the HTTP client and server programs and the underlying network performance.

Table 5 shows the average time for several exemplary query operations. These numbers were collected by invoking the MLA programs from the command line, using the same method by which they are invoked as CGI applications (i.e. passing arguments through environment variables). The results of the query were sent to the null device; e.g.,

```
mllaquery -h 1000 >/dev/null
```

Test runs were repeated 10 times each; the reported times are the average of these runs. No other queries were done in between so subsequent queries should find much of the information in the system buffer cache. Using a warm cache is representative of normal usage; if users do repeated queries they are typically done to narrow the hit set.

The test query operations are:

- 1000 items: return the most recent 1000 messages in the archive, no constraints are applied
- 5000 items: the same as 1000 items, but return five times as many messages

| Test Query | Hits | Time  |        |       |
|------------|------|-------|--------|-------|
|            |      | User  | System | Real  |
| 1000 items | 1000 | 0.85  | 0.07   | 0.95  |
| 5000 items | 5000 | 16.80 | 0.42   | 17.06 |
| date range | 200  | 0.08  | 0.04   | 0.12  |
| match 1 re | 200  | 0.44  | 0.06   | 0.53  |
| match 2 re | 5    | 0.49  | 0.06   | 0.57  |

Table 5: Sample MLA query tests using the majordomo archive. All times are in seconds.

- date range: return the first 200 messages between 1/1/95 and 1/1/96
- match 1 re: constrain the date range query to messages with "BOUNCE" in the subject
- match 2 re: constrain the match 1 re query to messages sent by users with "Brent" in their name (case sensitive)

Constraining a query to a date range is fast because the TOC includes table of messages sorted by date. Any other constraints require mlaquery to apply a regular expression package to header strings that are taken from the string pool. This results in an increase in the query time that is typically proportional to the length of the strings that are checked. Note that mlaquery first applies any date range constraint before doing regular expression matching so any cheap culling of the hit set is done before the more expensive pattern matching.

One should note the 5000 items test result; the time to carry out this query is not a linear product of the number of items to return. This is because the HTML document generated by the query is not proportional in size because the encoded hit set returned in the URLs is significantly larger (due to the encoding scheme). The running time of mlaquery is dominated not by the time to do the query but instead by the time to generate the resulting HTML document. This is an example of the limitation of the scheme by which query context is stored in the client.

## 4.0 Summary and Future Work

The Mailing List Archive (MLA) tools provide an efficient system for storing and retrieving mailing lists. The ability to access and navigate archives via the World Wide Web is valuable; providing high quality graphical user interfaces basically for free. The underlying database design for the table of contents and message bodies has worked out well. The scheme for storing query context on the client side of an HTTP connection has proven successful so long as the hit set size is constrained to be reasonably small (several hundred).

The design and implementation of the MLA tools was done in early 1994. At that time, they were intended mainly to support a few private mailing lists; since then they have been used in a variety of settings with good results. The two main enhancements that have been requested are: support for searching multiple archives and searching the body of mail messages.

The first request—to search multiple archives—is contrary to the original design. In most cases, what people really want is not the MLA tools but the user

interface and the ease of access that are provided by the tools.

The second request for searches over the body of mail messages has been added in an experimental version of the tools. Support was added to automatically construct inverted keyword indices from the entire text of each mail message. This new functionality permits queries to be formulated that search more than just the headers of a mail message. However, the indices are larger than desired (with only minimal filtering of uninteresting words they take up about equal space to the compacted message bodies) and compacting them opens up the usual set of issues: one can index fewer words by applying relevancy information to weed out unimportant information, or one can use data structures that support two-level queries such as those used in glimpse [2]. The indexing support was added in a way that makes it easy to evaluate alternate indexing techniques. This area is the subject of ongoing work.

## 5.0 Bibliography

- [1] Kevin Hughes. Hypermail 1.02. <http://www.eit.com/goodies/software/hypermail/hypermail.html>.
- [2] Udi Manber, Sun Wu. "GLIMPSE: A Tool to Search Through Entire File Systems". TR 93-34. Department of Computer Science, University of Arizona. October 1993. <http://glimpse.cs.arizona.edu:1994/>.
- [3] The RAND MH Message Handling System. UCI Version 6.8.3. October 29, 1995.
- [4] NI, Usenet search gateway by Mike Burrows.
- [5] Stephen R. van den Berg. Procmail & formail mail processing package. <ftp://ftp.informatik.rwthachen.de/pub/packages/procmail/procmail.tar.gz>.
- [6] D. Crocker, "Standard for the format of ARPA Internet text messages", August 13, 1982. Updated by RFC1327, RFC0987. <ftp://ds.internic.net/rfc/rfc822.txt>.
- [7] T. Berners-Lee, R. Fielding, H. Nielsen, "Hypertext Transfer Protocol -- HTTP/1.0". May 17, 1996. <ftp://ds.internic.net/rfc/rfc1945.txt>.
- [8] L. Deutsch, "DEFLATE Compressed Data Format Specification version 1.3". May 23, 1996. <ftp://ds.internic.net/rfc/rfc1951.txt>.
- [9] Margo Seltzer, Ozan Yigit. "A New Hashing Package for UNIX". USENIX Technical Conference. January 1991. Dallas, Texas. pp. 173-184.
- [10] Jean-loup Gailly, Mark Adler. zlib data compression library.

<ftp://ftp.uu.net/pub/archiving/zip/zlib/zlib0.99.tar.gz>.

- [11] Digital Equipment Corporation. "AltaVista Search". July 21, 1996.  
<http://altavista.software.digital.com/products/search/whitpaper/>.
- [12] NCSA. "The CGI Specification, Version 1.1".  
<http://hoohoo.ncsa.uiuc.edu/cgi/interface.html>.
- [13] Deja News. <http://www.dejanews.com/>.

**Sam Leffler** is a member of the corporate research group at Silicon Graphics where he works on a variety of projects. His current focus is on processor scheduling and operating system support for high performance multi-threaded applications. Previous work has spanned the gamut from 3D graphics to computer networking to programming languages. He received an M.S. in Computer Science and a B.S. in Mathematics from Case Western Reserve University, both in 1980.

**Melange Tortuba** is president of Tortuba Consulting, a small thinktank located in Berkeley California. She spends most of her time researching the needs and activities of overly-pampered domesticated felines. Her most well-known accomplishments to date have been in non-computer fields. She was educated at the University of California at Santa Barbara where she majored in mooching attention from students and faculty.



# Experiences with GroupLens: Making Usenet Useful Again\*

Bradley N. Miller  
John T. Riedl  
Joseph A. Konstan

*Department of Computer Science*  
*University of Minnesota*  
email: {bmiller,riedl,konstan}@cs.umn.edu

## Abstract

Collaborative filtering attempts to alleviate information overload by offering recommendations on whether information is valuable based on the opinions of those who have already evaluated it. Usenet news is an information source whose value is being severely diminished by the volume of low-quality and uninteresting information posted in its newsgroups. The GroupLens system applies collaborative filtering to Usenet news to demonstrate how we can restore the value of Usenet news by sharing our judgements of articles, with our identities protected by pseudonyms.

This paper extends the original GroupLens work by reporting on a significantly enhanced system and the results of a seven week trial with 250 users and over 20,000 news articles. GroupLens has an open and flexible architecture that allows easy integration of new news-reader clients and ratings bureaus. We show ratings and prediction profiles for three newsgroups, and assess the accuracy of the predictions.

## 1 The Problem with Usenet Today

### 1.1 Problem Statement

The information super-highway promises to deliver more information more rapidly than was

ever before possible. However, many of us are already overwhelmed with the amount of information we must process each day. The problem of information overload leaves us unable to keep up with the information we need. To long-time readers of Usenet news this problem is especially evident and has caused many users to abandon Usenet altogether. How did we get into this predicament?

The Internet was born in the 1970's by a group of like-minded scientists who used the net primarily to serve the interests of research and academia. This community of on-line pioneers thrived for about 20 years until the rush to the net began in the early 1990s. With this rush came millions of new users with interests that went way beyond the research questions that tied the early community together. Not only did the new users increase the volume of information on the net, they fundamentally changed the culture. What once felt like a small community now feels like a loud impersonal city.

The current estimate of Usenet volume is 21 million users posting 130,000 articles per day. This is up from an estimate of 10,000 articles per day in January 1994. The growth of the Web is even more phenomenal with current estimates that the size is doubling every 4 months, in terms of both traffic and the number of sites.

The GroupLens project seeks to alleviate the problem of information overload by applying collaborative filtering techniques to Usenet news and other Internet resources. In so doing we hope to help restore order to Usenet,

---

\*Thanks to AT&T Research for their generous support.

and build a renewed sense of community. In an earlier paper [10] we reported on the initial GroupLens architecture and a small scale pilot test with approximately 12 participants at our local universities. In this paper we report on the new GroupLens architecture, including the newly published open protocols, and a larger scale Internet wide user test involving more than 250 participants and tens of thousands of ratings. The next sections describe some of the past and current strategies for fixing Usenet. Section 2 presents the GroupLens server architecture. Section 3 discusses the client library and the adaptation of newsreaders to support GroupLens. Section 4 presents data gathered and lessons learned from the seven week user trial, and section 5 discusses future work and presents some conclusions.

## 1.2 Non-collaborative Solutions

Since the early days of Usenet people have tried to find ways to reduce the number of messages they must process each day. In this section we consider *non-collaborative solutions*, which are solutions that use only information from a single user. Some of the earliest non-collaborative techniques relied on matching keywords in the header fields of the news articles. Later, more sophisticated keyword matching techniques emerged that applied techniques from information retrieval.

**Kill files / Score files** One of the first methods introduced to the Usenet community to reduce the noise level was the killfile. Recently scorefiles have been introduced as a more general mechanism. A killfile allows a user to specify certain subjects or authors that he never wants to see, while a scorefile allows the user to give interesting subjects and authors high scores and uninteresting subjects and authors low scores [5]. The problem with these techniques is their coarseness. Not all articles containing a desired keyword are interesting, and even generally poor writers occasionally produce an article worth reading. Additionally, keywords are difficult to identify in the

presence of aliases, synonyms and misspelled words.

**Moderated Newsgroups** Another approach to reducing the noise level on Usenet is the creation of moderated newsgroups. In a moderated newsgroup one person, the moderator, must approve each article before it is distributed throughout Usenet. The moderator is responsible for rejecting articles that are off-topic, inflammatory, or generally of poor quality. The problem with moderated groups is that they require a large time commitment from the moderator, and the quality judgment is left up to a single person.

**Programmable Agents** Programmable agents are simple programs that perform actions on behalf of users. Programmable agents have been used in information filtering to prioritize messages, gather messages into folders by keyword, or even reply to messages. For instance, the Information Lens system enables even unsophisticated users to automatically perform actions in response to messages [6]. Object Lens extends the Information Lens to other domains, including databases and hypertext [3].

**Intelligent Agents** The July 1994 issue of *Communications of the ACM* is devoted to the state of the art in intelligent agent research. This issue includes discussions of several agents designed to reduce information overload. [4]. The agents address meeting scheduling, email handling, and netnews filtering. The netnews agent is known as NewT [14]. A user trains NewT by showing it examples of articles that should and should not be selected. The agent performs a full text analysis of the article using the vector-space model [11]. Once the agent has gone through initial training it starts making recommendations to, and accepting feedback from the user. Based on user feedback NewT is able to make weighted judgments about news articles containing keywords. Intelligent agents for information filtering suffer

from the same drawbacks as keyword based techniques. An additional problem is that agents must be trained. Norman points out in [9] that interaction with and instruction of agents is a difficult problem that has not been solved satisfactorily.

### 1.3 Collaborative Solutions

*Collaborative filtering* systems make use of the reactions and opinions of people that have already seen a piece of information to make predictions about the value of that piece of information for people who have not yet seen it. Collaborative filtering is already used heavily in informal ways. Users regularly forward articles, or references to articles, to their friends and colleagues with the explicit or implicit message: "You will like this." Collaborative filtering systems attempt to formalize this process to more effectively incorporate a larger set of users and data. The utility of collaborative filtering extends beyond the domain of Usenet news into the realm of movies, videos [2], and audio CDs [13].

Usenet represents a uniquely challenging problem for a collaborative filtering system because of the sheer volume of information items for which ratings must be collected and predictions calculated. The 130,000 new messages produced on Usenet each day dwarfs the number of new CDs and movies produced in an entire year.

**Tapestry** The Tapestry system [1] is an early collaborative filtering system designed to help small groups of people work together to solve the information overload problem. Tapestry makes sophisticated use of subjective evaluations. It allows filtering of all incoming information streams, including email and Usenet news. Many people can post evaluations and users can choose which evaluators to pay attention to. The evaluations can contain text, not just a numeric rating or boolean accept/reject. In the Tapestry system users can combine keyword criteria, along with subjective criteria to form requests. An exam-

ple request might be "Give me all the articles containing the word *collaborative* that Pat has evaluated and where the evaluation contains the word *excellent*." Tapestry works well in a close-knit community with common interests. GroupLens extends the concepts in Tapestry system in two ways. First, GroupLens provides predictions based on the aggregation of ratings entered by other users. Second, GroupLens does not require the user to know whose evaluations to use in advance.

**NoCeM** NoCeM (no see 'em) is a system that makes it possible for anyone to attempt to cancel an article that is widely cross-posted or seen as a blatantly commercial posting. Under the NoCeM model, any person on the net who sees something they think shouldn't have been posted can issue a NoCeM notice. However, just as with any other type of Usenet message, the weight the notice carries will be no greater than the poster's net.reputation. If people agree with the issuer's criteria and also feel that this person is a good judge of that standard then they will accept his/her notices. When a NoCeM notice is accepted by a user it will typically mark the message as read in the user's newsrsrc file. NoCeM notices could instead be used to remove the message from the local spool, thus keeping all users on the local system from seeing the article.

### 1.4 The GroupLens Approach

GroupLens is a collaborative filtering system for Usenet news. The aim of GroupLens is to help people work together to find articles they will like in the huge stream of available Usenet articles. In effect, GroupLens automatically selects for you a group of people to act as your personal moderators for a given newsgroup. These moderators are selected by finding people with whom you have had substantial agreement on past articles. Users can ensure their privacy by entering ratings under pseudonyms without reducing the effectiveness of the predictions.

Usenet news readers can take advantage of

GroupLens by reading news with a GroupLens-aware news client. We provide several clients and make a library available for newsreader authors who want to make their newsreaders GroupLens-aware. An example client is shown in figure 1. The newsreader connects to the user's local NNTP server to retrieve Usenet news articles, and it also connects with the GroupLens server to share filtering information. Whenever the user fetches articles from a newsgroup, the news reader sends a message to the GroupLens server requesting *predictions* of how the user will value each article. Those predictions are displayed alongside the article titles as a bar. When reading news articles, the user may enter ratings of the actual value of each article. Those ratings are sent back to the GroupLens server to serve as input for other users' predictions and to update the correlations between this user and other users. The more one uses the system, the more data there is upon which to base predictions.

We believe that GroupLens provides the best opportunity for managing the overwhelming amount of data in Usenet news. In addition to the general benefits of collaborative filtering over non-collaborative solutions, GroupLens has a scalable open architecture that can support a large number of users and data elements. The GroupLens architecture also can support a variety of algorithms for collaborative filtering, allowing system designers to trade off between efficiency of calculation, storage requirements, and the degree of personalization of predictions.

## 2 The GroupLens Architecture

### 2.1 Overview

At the heart of GroupLens lies the GroupLens Ratings Bureau (GLRB) which functions as a request broker for the distributed collaborative filtering engine. To implement the request broker we have adopted a "process pool" model that allows the GLRB to identify incoming re-

quests and hand them off to the appropriate background daemon. To keep the number of network connections low we have adopted a virtual session model for each client connection. Once a client is logged in the client is given a session token that is valid until the client logs out or becomes inactive.

In figure 2 we see two newsreading clients in different stages of communication with GroupLens. The *tin* client is in the process of establishing a connection with the GLRB; its request has not yet been determined. On the other hand the GLRB has already assigned the *xrn* client to the appropriate prediction daemon.

The filtering engine contains four primary modules. A prediction module, a ratings module, a correlation module, and a data management module. The GroupLens architecture is open, and the modules communicate with each other through a well defined protocol. This allows any of the modules to be replaced by a functionally equivalent module so long as the new one conforms to the protocol.

The primary goal of the collaborative filtering engine is to provide clients with accurate predictions quickly. Predictions are calculated by one of the daemons in the prediction daemon pool. To calculate a prediction for an article the prediction daemon requires two inputs: a measure of similarity between pairs of users, and ratings for the article in question. The way in which these two inputs are combined for each prediction is described in [13, 10, 7]. Our performance goal for the prediction algorithm is to be able to calculate and deliver 100 predictions in less than two seconds. In practice we are able to deliver 100 predictions in 4.2 seconds on a Sparc 5 workstation with 32Mb memory, running Solaris 2.4.

Pairwise similarity between users is determined by the correlation program. Similarity between two users is determined by how they have rated articles in the past. Because user's correlations change relatively slowly, and because newsreading tends to be a daily activity, the correlation program is run once a day.

Ratings for Usenet articles are received in

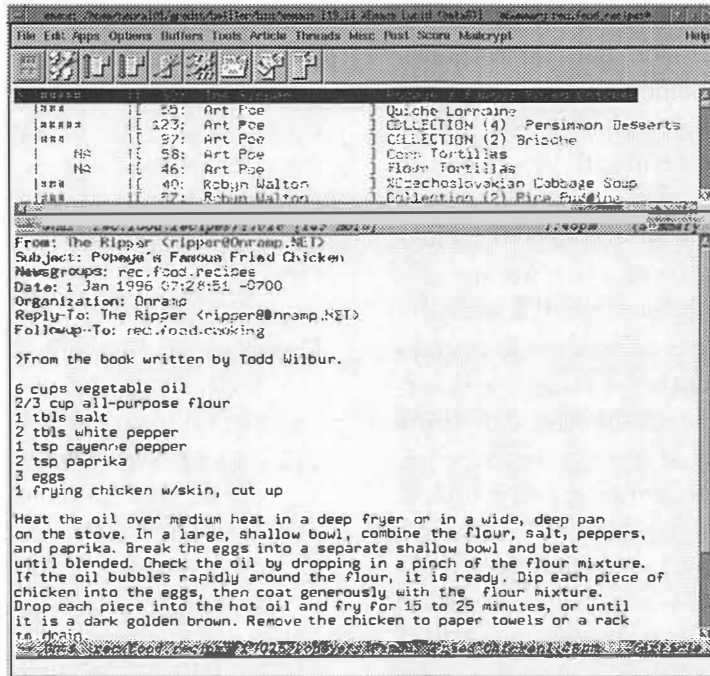


Figure 1: The GNUS newsreader with GroupLens predictions

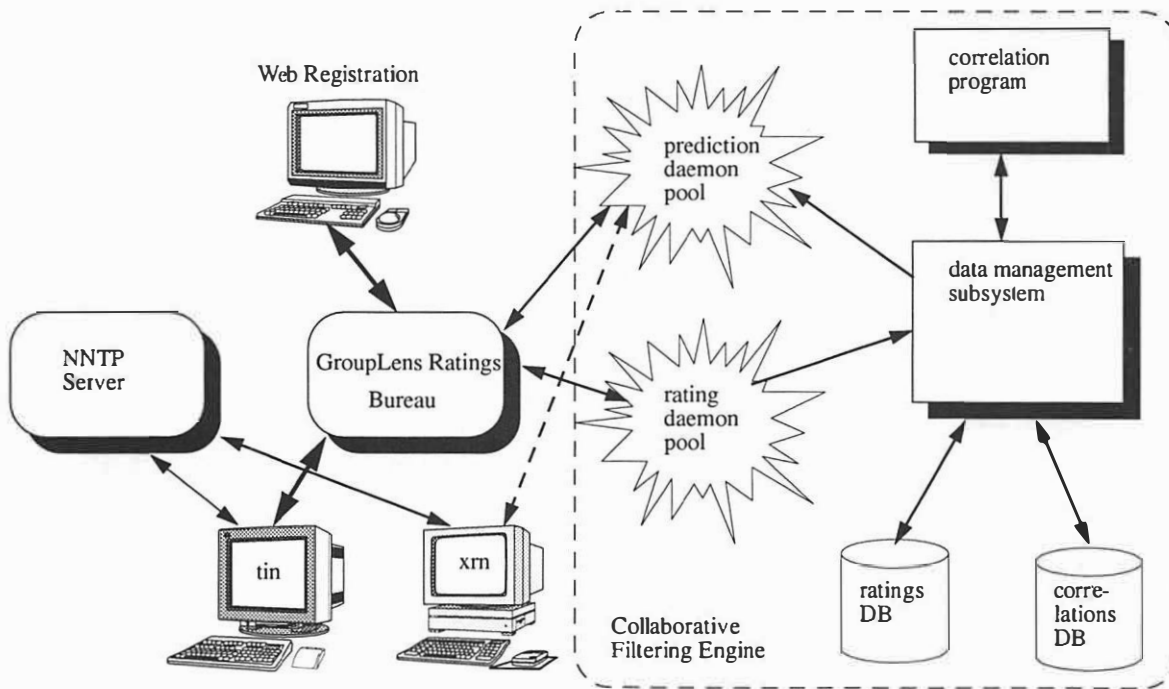


Figure 2: GroupLens Architecture Overview

batches from the newsreader client as defined by the GroupLens protocol. It is the responsibility of the ratings daemon to receive ratings from the clients as fast as possible and to ensure that the ratings are eventually stored in the ratings database. Our performance goal for ratings is to be able to accept 100 ratings in less than 1 second. In practice we are able to accept 100 ratings in less than 0.5 seconds.

The data management subsystem is responsible for maintaining both the ratings and correlations databases. Logically you can think of the ratings database as a large matrix organized with message-ids indexing the columns, and pseudonyms indexing the rows. The current database for the rec.humor newsgroup has 20,837 columns and 74 rows for a total of 1,541,938 cells. However, only 30,537 or 0.01% of these cells are occupied. We have devised a storage mechanism that minimizes access time for an individual cell, and minimizes the space required to store the matrix.

We have designed our data management interface so that we can plug in one of several DBMSs on the back end, while maintaining a consistent interface on the front. Currently we support three back ends: gdbm [8], Illustra, and OBST.

## 2.2 Protocol

The glue that ties all of the modules together, and allows the newsreading clients to talk to the GLRB and other modules, is the GroupLens protocol. The protocol consists of five major commands. We'll give a brief overview of the major commands here. The details of the protocol are available on-line [15].

Three key concepts in the protocol are *pseudonyms*, *tokens*, and *message-ids*. Pseudonyms are the secret identifiers selected by users to identify themselves to the GroupLens system while maintaining their privacy. Tokens are integers returned from the server to represent the state of a logged in user. The server maintains just enough state for each token so the user does not have to authenticate herself each time she

requests predictions or submits ratings. The server discards tokens after a timeout period. Message-ids are used by the clients to identify items they wish to rate or get predictions for. Message-ids in the Usenet trial are the standard Usenet message identifiers used by news clients to identify messages to the news server.

**Register** In figure 2 we see a World Wide Web client talking to the GLRB from the GroupLens Registration page. The format of a register command is `register pseudonym`. When the GLRB receives a register command, it checks the user database to make sure that the given pseudonym does not already exist.

**Login** The format of the login command is `login pseudonym`. When a login request is received the GLRB checks to see if the pseudonym is valid. If the pseudonym is valid the client is given a session token. To increase security a password may be optionally supplied as part of the login command.

**Logout** The format of the logout command is `logout token`. When a logout command is received the token is removed from the list of active tokens, and the token number is invalidated. Any future requests using this token number will be refused.

**GetPredictions** The format of the `getpredictions` command is `getpredictions token newsgroup`, followed by a list of Usenet message-ids. When the GLRB sees that the request is `getpredictions` it validates the token and newsgroup name, and then passes the request to a free prediction daemon. The prediction daemon reads the list of message-ids and returns either a prediction or a keyword indicating no prediction for each message-id.

**PutRatings** The format of the `putratings` command is `putratings token newsgroup`, followed by a list of tuples

that contain the message-id and rating. The ratings daemon simply reads the list of tuples and informs the client that it has received them. When it has time, the ratings daemon writes the list to the database.

## 2.3 Privacy

Privacy is an important issue in a large scale collaborative filtering system. There are three ways to handle user privacy issues in a collaborative filtering system. First, users may be anonymous so that ratings are submitted without any user identification. When ratings are submitted anonymously, the only operations the system can perform are aggregate operations such as the average rating [7]. Second, users may be known to all other users. In this case the ratings are closely associated with the reputation of the rater, and users seeking recommendations or predictions may specify which other users to use in generating predictions [1]. This option requires users to give up the privacy of their ratings. The third option, employed by GroupLens, uses *pseudonyms* to uniquely identify every user. Using pseudonyms allows ratings to be associated with a user, and allows predictions to be customized for users based on their correlation with other pseudonyms. In GroupLens, pseudonyms and their associated ratings are publicly available. However, we do not associate these pseudonyms with a user's real identity and we use an authentication protocol [12] that prevents a user from using another's pseudonym.

## 3 Filtering Clients

### 3.1 News Readers

The primary user interface for the GroupLens system is a set of newsreaders that are adapted to use the GroupLens server as well as the local NNTP server. We currently support three Unix-based newsreaders: *xrn*, *tin*, and *gnus*.

We are in the process of adapting newsreaders for the PC and Macintosh platforms.

To simplify the process of adapting newsreaders, we have implemented and freely provide a GroupLens client library. This library handles all GroupLens server communication, manages local configuration files, and provides data structures to simplify the integration of ratings and predictions into an existing news reader. When using the library, the newsreader author is freed from the details of logging into the GroupLens server and maintaining a token, and from the details of the GroupLens protocol.

Adapting a newsreader to use GroupLens involves three steps:

1. Passing the set of article IDs to the client library (to retrieve predictions) when retrieving article headers for a newsgroup.
2. Recording article information, including user ratings, in the GroupLens article table, and calling the library routine to submit these ratings after finishing each newsgroup.
3. Defining a user interface for displaying predictions (some support is in the library) and for receiving ratings from users.

Our experience modifying newsreaders has shown that the interface changes are the hardest part of the process. Many newsreaders use nearly every key on the keyboard, and consequently require creative interface design to maintain a consistent interface. While displaying predictions was somewhat simpler, we have found that some news reading models are not as amenable to selection by title and prediction, and accordingly plan to investigate methods for providing summary predictions for threads (perhaps at the client interface).

We were able to add GroupLens support to *xrn* with less than 1000 additional lines of code. These lines represent less than 3% of the total *xrn* source code.

### 3.2 Filter Bots

We use the name *filter-bot* to refer to simple filter programs that algorithmically (“robotically”) supply useful information to a filtering system. In GroupLens, a filter-bot is a program that assigns a rating to a Usenet article based on some simple computable criteria. Filter-bots are implemented using the client library and enter ratings under their own unique pseudonyms. This allows real users to weigh the ratings of the filter-bots along with ratings from other users. Examples of filter-bots we have implemented include an article length filter-bot and an excessive quoted text filter-bot. We intend to implement several more including a reading level filter-bot<sup>1</sup>, a prolific author filter-bot, and an excessive cross-posting filter-bot.

Filter-bots measure syntactic features of articles, providing additional ratings with which users can correlate, potentially improving predictions. Filter-bots facilitate incorporating new information filtering algorithms into the GroupLens architecture. Filter-bots also mitigate the *first rater problem*, which stems from the fact that in order to compute a prediction for an article at least one previous rating must be available. Filter-bot ratings for an article can be computed immediately, so they are always available for users.

## 4 Experiences

We now turn to the results of a user trial we conducted to test the GroupLens architecture. The user trial began February 8, 1996 when we posted an announcement to the `comp.os.linux.announce` newsgroup. In order to participate in the trial, users had to be willing to use one of the newsreaders that had been enhanced with GroupLens support. The newsreaders available were `gnus-5.1` (for `emacs`), `tin`, and `xrn`. Participants also had to be willing to read and

<sup>1</sup>A reading level filter-bot rates articles according to the minimum grade level for which the vocabulary and sentence structure would be appropriate.

rate articles in one of our supported newsgroups. The groups supported for the trial included the entire `comp.os.linux` hierarchy, `rec.humor`, `rec.food.recipes`, `comp.lang.c++`, `comp.lang.java`, and `rec.arts.movies.current-films`. Participants were told to rate articles according to the following definitions:

1. This article is really bad! a waste of net.bandwidth.
2. This article is bad.
3. This article is neither good nor bad.
4. This article is good.
5. This article is great, I would like to see more like it.

Through the first seven weeks of the trial, we had 250 users register to use GroupLens. The total number of ratings received during the seven week period was 47,569. These ratings are spread over a total of 22,862 distinct messages. Over the same seven week period GroupLens provided over 600,000 predictions to users. This ratio of ratings to predictions is appropriate for a noisy domain like Usenet news, since it suggests that ratings help users choose which items to review.

In the next two sections we take a look at the general question, “does it work?” We’ll look at this question from two perspectives: First, do the predictions accurately reflect what the user rated the article? Second, do users find the predictions useful, and do they believe them? We used data collected from the earlier GroupLens trial [10] to develop prediction algorithms, and evaluated the performance of the prediction algorithms based on the data from the present trial. The algorithms were not changed during this trial.

### 4.1 Accuracy of Predictions

Our experience has shown that the prediction program behaves differently for different newsgroups. To study this point, we

will examine the accuracy of the prediction program for three representative newsgroups. `rec.humor`, `rec.food.recipes`, and `comp.os.linux.development.apps`

For each of the newsgroups we will compare the accuracy of predictions for two different ways of calculating the predictions. We will calculate a personalized prediction for each user for each article using the Pearson coefficient as a similarity measure between users, as described in [10]. For comparison we will calculate the average rating entered for each article. We compare each prediction against the actual ratings entered by the users. It is useful to look at the average because it is fast to calculate and requires very little storage. On the other hand, the average does not allow for any personalization of the ratings.

The metrics we will use to measure the accuracy of the algorithms include the *mean squared error*  $\overline{E^2}$ ; The *mean absolute error*  $|\overline{E}|$ ; the *standard deviation*,  $\sigma$ , of  $|\overline{E}|$ ; and the Pearson correlation coefficient  $r$  between ratings and predictions.

To measure the mean absolute error we take the absolute value of the difference between the actual rating entered by the user and the prediction computed by the algorithm for each rating/prediction pair, and compute the mean of all of the differences. The lower the mean absolute error, the better the algorithm. The standard deviation of the error is a measure of how consistently accurate the algorithm is. One problem with using the mean absolute error is that it does not sufficiently penalize algorithms that make large errors. The mean squared error, like least squares regression, disproportionately penalizes algorithms that make large errors more than small. We want to penalize large errors because users probably don't distinguish between a prediction of 1.5 and 2.0. On the other hand, users will notice if an algorithm predicts something to be a 4.0 that should really be 2.0.

As mentioned in section 2, the prediction algorithm requires a measure of similarity between pairs of users (correlation), and ratings. The nature of the newsgroup appears to have

an effect both of these factors. In figure 3 we show the rating profiles for all groups combined, and for the three newsgroups.

In `rec.humor`, 83% of the ratings are 1 or 2. This reflects the paucity of funny articles and the overabundance of name-calling, flaming, and completely silly discussions of World War II. `Rec.humor` is a good example of a newsgroup where there is a clear metric for determining a rating: "Is it funny?" The fact that there is a clear metric for judging each article, and the fact that there is so much noise leads to a high level of correlation between pairs of users. This is illustrated in figure 4 where we can see that most pairs of users have a high positive correlation.

| method       | $\overline{E^2}$ | $ \overline{E} $ | $\sigma$ | $r$  |
|--------------|------------------|------------------|----------|------|
| average      | 1.1              | 0.63             | 0.88     | 0.49 |
| personalized | 0.94             | 0.67             | 0.68     | 0.62 |
| all-ones     | 2.01             | 0.78             | 1.18     | NA   |

Table 1: Summary of Results in `Rec.humor`

In table 1 we see the comparison between average and personalized predictions for `rec.humor`. Because there is such a high degree of correlation between users, we see that the average is slightly better than the personalized algorithm in terms of the mean absolute error. One might think that given the ratings profile for `rec.humor` the best strategy to minimize error would be to simply predict 1 for every article. The row called "all-ones" in table 1 shows that this is not a good strategy after all.

In `comp.os.linux.development.system`, and `rec.food.recipes` we see that the ratings are more evenly distributed (see figure 3). `Rec.food.recipes` is a moderated newsgroup, so all of the posts are on topic, and there is no name calling or spamming. In addition users once again have a clear metric for rating an article: "Would I like to cook this?" However, as figure 4 shows, users in `rec.food.recipes` have a lower correlation. The reason for this is that ratings are based literally on taste. For

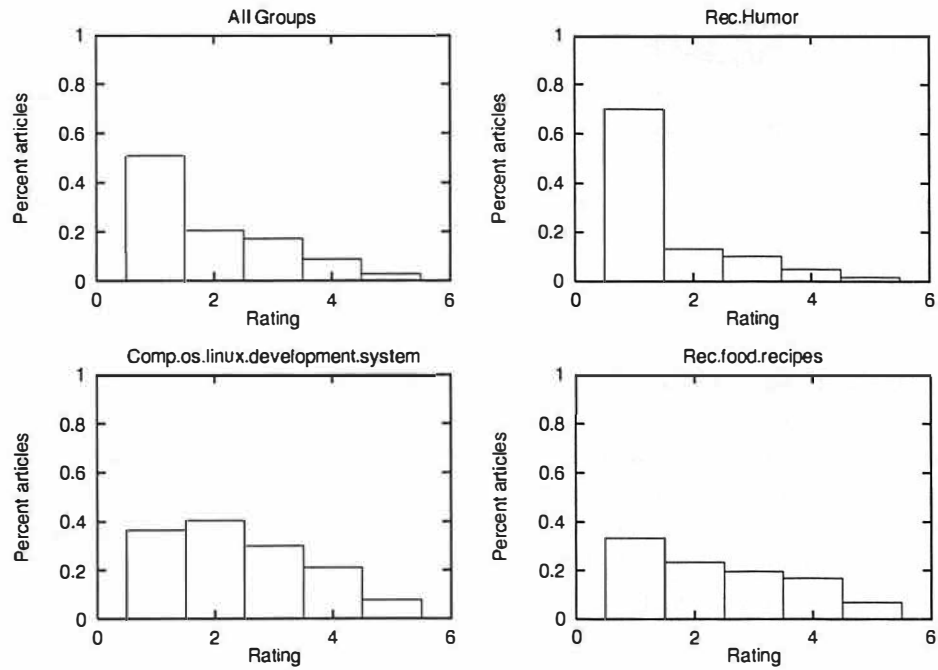


Figure 3: Summary of ratings in selected newsgroups

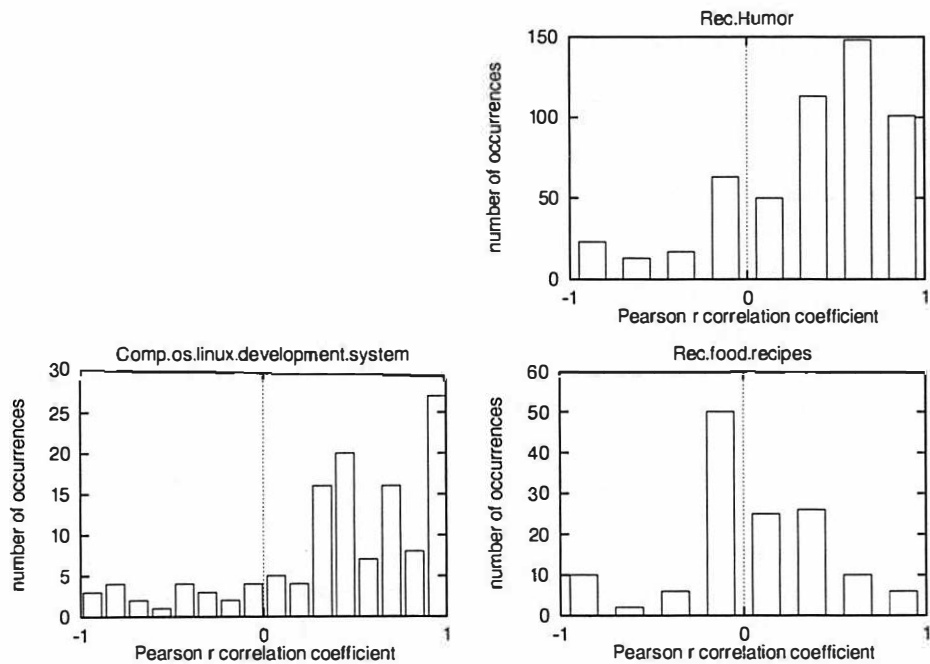


Figure 4: Summary of correlation between users

example two users may agree all the time on desert recipes, but one may be a vegetarian who rates all recipes with meat low, and the other may be a carnivore who rates recipes with meat high.

In table 2 we summarize the results for `rec.food.recipes`. We see that the errors in this group are uniformly higher than for `rec.humor`. This can be attributed to the fact that the correlation between users is low for this newsgroup. However, we do see that the personalized predictions are better than average predictions for all of our error metrics.

| method       | $E^2$ | $ \bar{E} $ | $\sigma$ | $r$  |
|--------------|-------|-------------|----------|------|
| average      | 2.65  | 1.29        | 0.97     | 0.05 |
| personalized | 1.94  | 1.09        | 0.86     | 0.33 |

Table 2: Summary of Results in `Rec.food.recipes`

Determining what to rate an article in `comp.os.linux.development.system` is more difficult than either of the two previous newsgroups. When rating an article in this group a user must weigh several factors:

- Is the article appropriate for this newsgroup?
- Is the topic of the article interesting to me?
- Is the article well written?
- Is the article factually correct?

Despite all of these factors the readers of `comp.os.linux.development.system` have a high degree of correlation. This may be because the early adopters of the GroupLens system are all likely to be fairly sophisticated linux users. In table 3 we see that the personalized predictions are again more accurate than the average.

## 4.2 Effect of Predictions on Users

We'll now look at what effect, if any, the predictions have on a user's likelihood to read and

| method       | $E^2$ | $ \bar{E} $ | $\sigma$ | $r$  |
|--------------|-------|-------------|----------|------|
| average      | 1.28  | 0.78        | 0.82     | 0.41 |
| personalized | 0.91  | 0.71        | 0.64     | 0.55 |

Table 3: Summary of Results in `Comp.os.linux.development.system`

rate a message. One result is clear: users are more likely to rate messages if they see that they are also getting predictions. Further, our analysis of the rating patterns of users shows that users are almost twice as likely to rate an article for which they see a prediction greater than three than they are for an article with a prediction of three or less.

## 4.3 Bootstrapping

One rather important lesson that we have learned over the course of this project is the difficulty of bootstrapping collaborative filtering in Usenet newsgroups. While we expected some inertia among users, and in particular recognized the difficulty in asking users to upgrade or change newsreaders, we were surprised by some of the social difficulties involved in bringing collaborative filtering to Usenet news.

Collaborative filtering must have users to be useful. In fact we believe that a collaborative filtering system has built in incentives that encourage more people to participate. To ensure value for trial users, we made an effort to add GroupLens support for newsgroups slowly, and only after ensuring that we had at least one or two active reader/raters who would keep the group going. We would then post a message to the newsgroup itself, inviting others to use GroupLens and pointing them to our web page for registration and software. It was here that we ran into a very tricky bootstrapping problem.

Discussing GroupLens was off-topic for almost every newsgroup we encountered. Accordingly, our messages were often ignored, and there was no follow-up discussion within the group. In retrospect, this is not surprising. A posting about a better way to read news

is not funny (and therefore does not belong in `rec.humor`), it is not about linux software and development (and therefore does not belong in `comp.os.linux.*`), and similarly is out of place in the very newsgroups where we expected it to be useful. Few Usenet news readers choose to read newsgroups *about* reading news.

We did get a user base, albeit more slowly than we would have liked. We have since recognized more effective ways of bootstrapping: working more closely with newsreader maintainers to become part of the standard distribution, providing limited service for a wider range of newsgroups to create more general interest, and direct promotion through demonstrations and other publicity.

## 5 Conclusions and Future Work

The GroupLens trial has demonstrated the efficacy of collaborative filtering for Usenet news. We have learned about the challenges of this vast domain. Each newsgroup brings forward new characteristics that affect the accuracy of our predictions. The sheer volume of Usenet news has forced us to have an efficient implementation. The numbers are in, and GroupLens provides value to participants. Anecdotal evidence supports this conclusion as we hear from users who long-ago abandoned the `rec.humor` newsgroup returning to it with GroupLens guiding them to a handful of funny articles in just a few minutes each day. Still, our work is far from finished.

There are many areas of future work to reduce the possible costs to users of using collaborative filtering. These costs come in three forms: (i) time spent entering a rating; (ii) performance costs incurred by the GroupLens software; (iii) the time wasted in reading articles that are predicted to be better than they really are.

One way of reducing the time spent entering a rating is to rely on implicit measures of interest, such as how long you spend reading an article, or whether you print or file the article

after reading it. Collaborative filtering studies are needed to compare the costs and benefits of explicit ratings with implicit ratings.

Performance costs for the current GroupLens system are low, but these costs increase with the number of users. One way to keep performance costs low is to develop distributed GLRBs that perform correlation and prediction independently on subsets of the user population. Scaling GroupLens to the Internet will require distribution, to keep communication and computation times.

Time wasted reading articles with high predictions that are actually uninteresting is difficult to control. In practice, predictions become increasingly accurate as more readers rate the the article. One way to make it easier for users to use collaborative filtering to select quality articles would be to develop prediction algorithms that include a confidence measure for the accuracy of the prediction. Confidence measures would help users make the tradeoff between opportunity cost and expected value.

GroupLens is an open architecture with freely distributable protocols. Anyone who wants to participate can write a news client to connect with our GLRB, or a new GLRB that offers improved service to our news clients. An open architecture encourages interaction and innovation by the community. For instance, one GroupLens participant has already written an proxy GLRB that downloads his ratings to Poland overnight so he gets better interactive performance!

The client library further encourages participation, by simplifying the task of integrating GroupLens with new news clients. For most news clients, only 2-3% of the code must be modified to support GroupLens. Recently, several of the maintainers of popular news clients have announced support for GroupLens. We look forward to working with the community to add GroupLens support to additional newsreaders. We also encourage the development of new filter-bots that use the client library to communicate computed ratings to GroupLens.

Usenet is on the one hand a rich and valuable information resource, and on the

other hand a quagmire of the useless and the tasteless. GroupLens lets us team up to drain the quagmire and separate the valuable from the useless. If we work together we can each peruse a fraction of the articles submitted each day, in exchange for having the interesting articles pointed out to us. More participants means more ratings available to GroupLens, which means even better predictions. The GroupLens experience is on-going at <http://www.cs.umn.edu/Research/GroupLens>. Join us in making Usenet useful again!

## 6 Acknowledgements

Many people have participated in making GroupLens a success. Paul Resnick deserves special recognition for co-founding the project with John Riedl. The HackWeek members – the authors, Jon Herlocker, Dave Maltz, and Paul Resnick – designed and implemented the new architecture. Danny Iacovou, Mitesh Sushak, and Pete Bergstrom worked on early versions of the system.

## References

- [1] D. Goldberg, D. Nichols, B.M. Oki, and D. Terry. Using collaborative filtering to weave an information tapestry. *Communications of the ACM*, 35(12):61–70, 1992.
- [2] W. Hill, L. Stead, M. Rosenstein, and G. Furnas. Recommending and evaluating choices in a virtual community of use. In *Proceedings of ACM Conference on Human Factors in Computing Systems (CHI)*, pages 194–201. ACM, 1995.
- [3] Kum-Yew Lai and Thomas W. Malone. Object lens: A “spreadsheet” for cooperative work. In *Proceedings of International Conference on Computer Supported Cooperative Work*, Portland, OR, September 1988. ACM Press, New York, NY.
- [4] P. Maes. Agents that reduce work and information overload. *Communications of the ACM*, 37(7):30–40, July 1994.
- [5] Lars Magne-Ingebritson. *Gnus 5.0 Reference Manual*. Available at: <http://www.miranova.com/gnus-man/gnus.html>.
- [6] T.W. Malone, K.R. Grant, F.A. Turbak, S.A. Brobst, and M.D. Cohen. Intelligent information-sharing systems. *Communications of the ACM*, 30(5):390–402, May 1987.
- [7] D.A. Maltz. Distributing information for collaborative filtering on Usenet net news. Master’s thesis, MIT Department of EECS, May 1994.
- [8] Philip A. Nelson. *gdbm-1.7 Reference Manual*. Available from: <ftp://prep.ai.mit.edu/pub/gnu/gdbm-1.7.3.tar.gz>.
- [9] D.A. Norman. How might people interact with agents. *Communications of the ACM*, 37(7):68–71, July 1994.
- [10] P. Resnick, N. Iacovou, M. Sushak, P. Bergstrom, and J. Riedl. Grouplens: An open architecture for collaborative filtering of netnews. In *Proceedings of Computer Supported Cooperative Work Conference*. ACM SIG Computer Supported Cooperative Work, 1994.
- [11] G. Salton and M. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, 1983.
- [12] Bruce Schneier. *Applied Cryptography*. John Wiley & Sons, 1994.
- [13] U. Shardanand and P. Maes. Social information filtering: Algorithms for automating “word of mouth”. In *Proceedings of ACM Conference on Human Factors in Computing Systems CHI*, pages 210–217, 1995.

- [14] B. Sheth and P. Maes. Evolving agents for personalized information filtering. In *Proceedings of 9th IEEE Conference on Artificial Intelligence for Applications*. IEEE Computer Society Press; Los Alamitos, CA, USA, 1993.
- [15] The grouplens home page.  
<http://www.cs.umn.edu/Research/GroupLens>.

# Overcoming Workstation Scheduling Problems in a Real-Time Audio Tool

Isidor Kouvelas      Vicky Hardman

*Department of Computer Science  
University College London*

{I.Kouvelas, V.Hardman}@cs.ucl.ac.uk

## Abstract

The recent interest in multimedia conferencing is a result of the incorporation of cheap audio and video hardware in today's workstations, and also as a result of the development of a global infrastructure capable of supporting multimedia traffic - the Mbone. Audio quality is impaired by packet loss and variable delay in the network, and by lack of support for real-time applications in today's general purpose workstations. A considerable amount of research effort has focused on solving the network side of the problem by providing packet loss robustness techniques, and network conscious adaptive applications. Effort to solve the operating system induced problems has concentrated on kernel modifications. This paper presents an architecture for a real-time audio media agent that copes with the problems presented by the UNIX operating system at the application level. The mechanism produces a continuous audio signal, despite the variable allocation of processing time a real-time application is given under UNIX. Continuity of audio is ensured during scheduling hiccups by using the buffering capabilities of workstation audio devices drivers. Our solution also tries to restrict the amount of audio stored in the device buffers to a minimum, to reduce the perceived end-to-end delay of the audio signal. A comparison between the method presented here (adaptive cushion algorithm), and that used by all other audio tools shows substantial reductions in both the average end-to-end delay, and the audio sample loss caused by the operating system.

## 1 Introduction

The ability of current wide area networks, such as the Mbone, to support multimedia conferencing has been recently demonstrated by a series of multicast events. One of the first such events was the Internet Engineering Task Force meeting on March 1992 [1]. The renewed interest in multimedia conferencing is a result of the provision

of audio and video hardware in UNIX workstations and the development of multicast for the Mbone [2]. Of the media used in multimedia conferences, audio is the most important [3], as speech is the natural mechanism of human communication, supplemented by video and shared text. Audio is also the only real-time service (video is normally slow-scan), which makes its provision at a reasonable quality of service (QoS) far more difficult.

The Internet's service model offers 'best effort' transmission, and is unable to provide the QoS guarantees needed for real-time traffic. As a result, audio quality is impaired by packet losses and variable transmission delays over the network, but it also suffers from the lack of support for real-time applications in general purpose UNIX operating systems. One approach to solving network problems has been to provide QoS guarantees through resource reservation [4]. Another approach has been to provide network conscious applications that adapt to the problems presented by the Mbone [5]. The latter mechanism has the advantage of being deployable without any network modifications.

Traditional time-sharing operating systems for general purpose workstations do not provide adequate support for real time applications [6]; they operate in an asynchronous fashion, and handle data in blocks [7]. Real-time audio applications are 'soft', in that they have to keep a continuously draining audio device driver fed with blocks of audio samples, but there is no specified instant when audio must be transferred, just a dead-line. However, the lack of a delay bound for timers and external events makes it impossible to guarantee a regular supply of audio samples. Current real-time audio applications ignore the problem, which results in frequently disrupted audio, and increased end-to-end delay.

A solution analogous to that of resource reservations to provide QoS guarantees is to modify the operating system scheduler to provide bounded dispatch latency for applications [7, 8, 9, 10]. Despite experimental proof that this approach can significantly improve the perceptual quality

of multimedia applications, it has not been implemented in the majority of desktop workstation operating systems.

There is a very large number of general purpose workstations connected to the Internet, that will soon want to use audio applications with the current scheduler. We present a solution at the application level, that smoothes out scheduling jitter in audio applications using the workstation's device driver buffering capabilities. The solution is analogous to playout adaptation for solving the network jitter, in that it does not require any modifications to workstation hardware or operating system.

This work is part of ongoing research at UCL to develop a robust and flexible audio conferencing component for use on general purpose workstations over the Mbone. Initially began as part of the MICE project [11], and later as parts of projects ReLaTe [12] and MERCI (Multimedia European Research Conferencing Integration, European Union Telematics Applications Programme #1007), the proposed solution has been implemented and evaluated in our Robust-Audio Tool (RAT) [5], which is now funded by an EPSRC project [13]. RAT provides real time audio connectivity between participants in multicast multimedia conferences over the Mbone. RAT is currently supported under SunOS, Solaris, IRIX, HP-UX, FreeBSD and Win32 and ports are under way for Linux PCs and Digital Unix platforms.

In this paper we analyse the audio scheduling problem and its implications. The solution and implementation in RAT is described, together with evaluation results that show the success of our method.

## 2 Background

The audio device can be visualised as two buffers:

- The input buffer continually inputs samples from the analogue audio input (microphone)
- The output buffer is fed with samples from the application and these are synchronously output to the loudspeakers.

The rate of input sample accumulation, and sample output is the same (sampling rate). The operating system buffer in the device driver has a fixed upper limit, and is adjustable under most OS platforms.

The audio application interacts with audio input and output through the two device driver buffers. Samples are read from the input buffer for processing and written out to the output buffer for playback. In contrast with actual audio playback, the transfer of blocks of samples between the audio application and the device driver is ad-hoc. Blocks are read from and written to the device driver buffers, and all audio processing takes place on this block unit size. The minimum block size is enforced by the

workstation's processing power; the larger the block size, the less frequently the application has to read and write audio blocks.

Figure 1 illustrates the basic functionality of our audio tool. The operation is similar to that of other existing conferencing audio tools like VAT [14], NeVoT [15] and IVS [16].

For the purposes of this paper the audio tool can be considered as a process that acts as an interface between the audio device and the network. The process provides a two way communication facility, and samples input from the audio device are processed and transmitted across the network to remote conference participants. Packets are received from the network, processed, and samples played out to the audio device. For a description of the functions involved in processing the audio samples and for more information on the structure of our audio tool see [5].

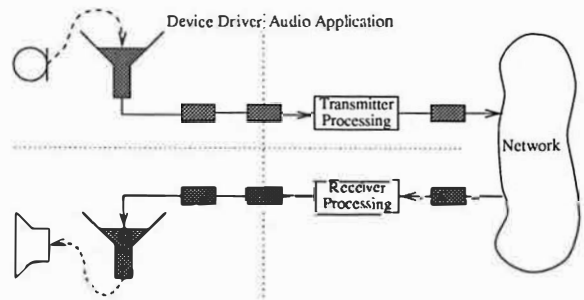


Figure 1: RAT operation

Mbone audio tools use silence detection and suppression to stop the transmission of audio when a conference participant is not active. The current algorithms operate by measuring the average energy in sample blocks [17]. A decision about speech/silence is made by comparing the average energy in a block against a threshold. Blocks that are judged to be above the threshold are labelled as 'speech', and transmitted. Blocks that are judged to be below the threshold are labelled as 'silence' and thrown away. This operation produces talkspurts (periods of continuous speech packets) inter-spaced with periods where no packets are transmitted. The threshold between speech and silence is adjusted during periods of silence.

The effect of sample loss on audio may produce significant degradation in the audio quality, depending on the length of individual gaps, and the frequency of occurrence. Audio gaps indicate a pause in speech to the human brain causing confusion and reducing intelligibility. This problem has been studied by the authors in [3, 5]. The minimisation of sample loss is a primary design goal in audio applications.

In real-time conferencing audio systems interactivity is known to be substantially reduced for round trip delays larger than 600 ms. Large round trip delays in conversational situations increase the frequency of confusions and amount of both double talking and mutual silence [18]. The components of this delay in a real-time Internet audio tool are:

- Processing and packetisation delay at transmitter.
- Variable delay experienced by audio packets traversing the network.
- Audio reconstruction buffering at the receiver necessary for smoothing out network delay jitter.
- Delay caused by queued audio samples in the operating system device buffers.

The minimisation of delay is also a primary design goal in audio applications.

### 3 Audio playback problem

There are two main goals when considering operating system effects for achieving good quality real-time audio communication:

- Continuous audio playback with no unnecessary breakups.
- Low delay in device driver buffers to minimise the end-to-end delay, so that interactivity and normal communication patterns are preserved.

To play-back audio, buffers of samples have to be transferred to the audio device driver. If all the samples in the device driver are played out before more are supplied, then the buffers run dry, and audio playback stops. To avoid the resulting gap of silence in the output audio, the transfer of samples must take place regularly.

Modern workstation audio device drivers provide a selection of audio sampling frequencies. Conferencing applications usually aim to transmit telephone quality speech, and consequently would like to use a sampling frequency of 8KHz. However, audio sampling frequency crystals usually do not have a nominal frequency of exactly 8kHz, and can vary considerably from one workstation to the next [19]. Timing events using the workstation clock without compensating for the drift in the audio crystal will lead to one conference participant's audio buffers becoming full, while a remote workstation's buffers may run dry. To simplify operations the sampling clock of the audio device is used instead. This is achieved by using the number of samples read from the audio device as an indication of the amount of time that has elapsed. Since one crystal is used for audio input and

playback, the number of samples read gives a very accurate count of the number of samples played out. With the accurate knowledge of the number of samples that have been consumed by the audio device, a receiver can calculate the drift from a transmitter and compensate by adjusting the duration of silence periods.

### 4 Implications for a Real-Time Audio Tool

Networked audio tools have a strict loop of operation in order to meet real-time timing restrictions. In each cycle of this loop the following basic operations take place:

- A block of samples is read in from the device driver input buffer.
- An equivalent number of received and processed samples are written out to the audio device.
- Other processing takes place which may involve packets being transmitted onto or received from the network.

Using this order of operations attempts to ensure that as many samples as needed are fed to the output buffers of the audio device. Feeding more would create a backlog of samples in the device driver to play out and would increase the delay. Providing less would result in gaps of silence in playback because of the buffers running dry.

On a non-multi-tasking machine, keeping up with this loop is not hard to achieve. Under a time-sharing operating system - like UNIX - this may not always be possible. The UNIX scheduler decides when a process gets control of the CPU. Processes are serviced according to their priority, and there is no useful upper bound on the amount of time a process may be deprived execution [7, 8].

As a result of other processes being serviced, a relatively large amount of time may elapse in between two instances of the audio tool being scheduled. If the time between schedules is larger than the length of audio data that was written last time, then an audible gap of silence will result in the output audio signal. The persistent effect of the gap in the audio output is to restrict the timing of the output, with respect to the input; extra delay is accumulated in the device driver, since each block of samples is played out later than it should have been. Measurements have shown that the accumulated interruptions caused by an intensive external event on a loaded workstation, can create a delay of several hundreds of milliseconds over the period of a cycle of operation of our program. Since the audio system is timed from the read operations of the audio device, the time that elapses when the audio tool process does not have control of the CPU will be evident; there is lots of audio in the input buffer of the audio device waiting to be read. In response to the waiting input

audio, the process will execute the loop several times in succession, until it reads the blocks in. For each of the blocks read, a block of equivalent size will be written out to the output buffers. (Figure 2) shows the delay increase diagrammatically.

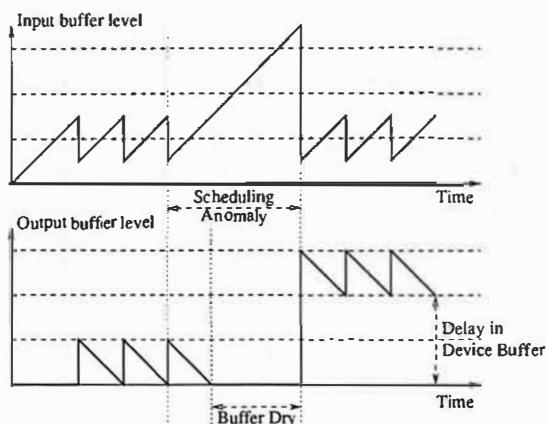


Figure 2: Accumulation of delay in output buffer

The extra buffered audio samples in the device driver prevent additional gaps, as there is now extra audio to play out while the audio tool process is not executing. Current Internet conferencing audio tools like VAT [14], NeVoT [15] and IVS [16] rely on this effect. During a talkspurt, the first CPU starvation causes a gap in the audio. Successive starvations will only cause a gap if they are longer in duration than the longest starvation so far. Consequently, the delay caused by the buffered samples in the audio device increases up to the length of the maximum interruption. At the end of the talkspurt the accumulated delay is zeroed, as transfer of samples to the audio device stops, and the audio device output buffer drains. If silence suppression is not enabled in the transmitting tool, then the increase in delay will persist throughout the duration of the session.

## 5 Adaptive buffer solution

In order to eliminate gaps in the audio signal, and minimise delay in the device driver buffer, adaptive control of the buffers is sought. The adaption algorithm will trade off the two variables. The adaption algorithm controls the amount of audio in the buffers, and ensures that there is always a minimum amount of audio to reduce the gaps caused by scheduling anomalies. Monitoring a history of the size of scheduling anomalies means that excessive audio is not buffered.

Intuitively, a situation where the amount of samples in the device driver buffer (called the cushion!) can cover for small frequent interruptions is ideal. This principle will not introduce excessive delay, but large infrequent

interruptions will still cause a gap in the audio signal. The size of the cushion must be determined by the current performance of the workstation, which can be estimated by analysing a history of scheduling anomalies. As new processes start, or external events happen, the overall perceived load and behaviour of the scheduler will change. In order to maintain the desired sound quality and minimise the delay, the cushion size must be adaptive, and should attempt to reflect the state of the workstation.

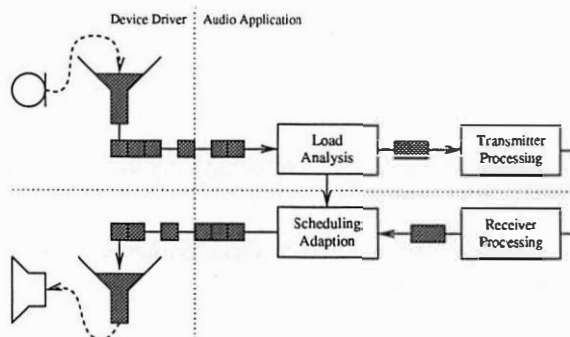


Figure 3: Adaptive cushion algorithm

### 5.1 Measuring workstation state

An estimate of the state of the workstation can be achieved by a simple modification to the structure of the audio application.

In the audio tool application loop, instead of reading a block of fixed size from the audio device driver, a non-blocking read is made, and all the stored audio is retrieved. The amount of audio gives an exact measure of the amount of time that has elapsed since the last time the call was made.

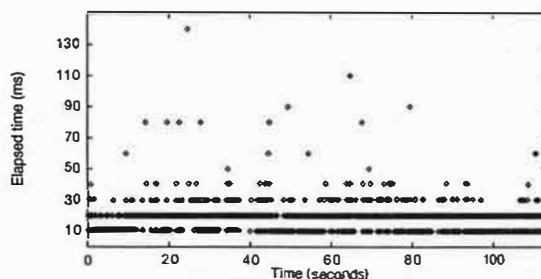


Figure 4: Read length variation

A history of the time between successive calls reflects the loading of the workstation. Figure 4 shows measurements from a lightly-loaded SUN Sparc 10 workstation over a period of two minutes. There are readings as often as every 10ms, which gives enough load samples to be able to monitor load changes as they happen.

There is an exception event to cater for in the implementation; when the amount of audio that is returned by the read call is equal to the total size of the device driver buffer. It is likely that an overflow has occurred, and some input samples have been lost, and therefore the track of time. The audio tool process resynchronises using external mechanisms like the workstation clock. This situation does not happen very often, since the total length of the device driver buffer is configurable, and is usually set large enough to cater for a few seconds of continuous audio input.

## 5.2 Cushion size estimation

Based on the workstation load information, the target fill level for the device driver buffer can be estimated.

Figure 5 shows the distribution of the measurements presented in figure 4. The X axis represents the elapsed time in milliseconds. Y axis is logarithmic and shows the number of times each different value occurred. It can be seen that the vast majority of measurements are smaller or equal to 40 milliseconds (320 samples). However the largest measurements are 130 milliseconds long.

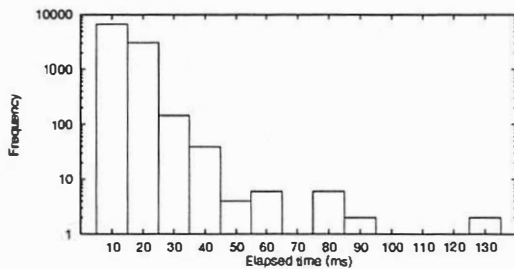


Figure 5: Load measurements distribution

The estimation algorithm should maintain a cushion level that will cater for the majority of scheduling anomalies, without resulting in gaps in the playback. This might be trivially achieved by maintaining output buffer levels at the maximum measured read length, but this will result in excessive delay being introduced. The adaptation algorithm introduced here allows the user to specify how much audio breakup should be traded for a reduction in delay.

The algorithm operates by maintaining a history of past load measurements. After a new measurement, the desired cushion size is estimated based on the recent history. The load measurements are stored in a circular buffer, and to avoid the processing overhead of examining all the logged data, a histogram of load measurements is maintained. The histogram is incrementally built, and each time a new load measurement is made, the oldest one in the circular buffer is removed from both the buffer and the histogram, and replaced with the new one. An esti-

mate is made by examining the histogram, and calculating a cushion value that will cover a given percentage of measurements.

Let  $b_i$  be the number of load samples of length  $i$  and  $h$  the length of the history. Then we have:

$$h = \sum_{i=1}^{\infty} b_i$$

If  $t$  is the threshold of read lengths which is being catered for, then the cushion  $c$  is given by:

$$c = \min x : \sum_{i=1}^x b_i > t$$

The cushion adaptation algorithm is configurable by adjusting the history length  $h$  and by varying the threshold  $t$  of load measurements that are going to be catered for.

The desirable behaviour would be to follow the trend in scheduling load, and avoid rapid jumps in cushion size due to very short-lived bursts. This can be achieved by increasing the history length, which in effect low pass filters the load information. The increase in the history length, however, is at the expense of fast adaption since the cushion estimate now depends on older data.

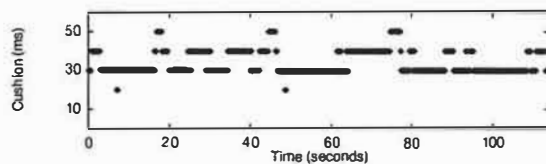


Figure 6: Fast adapting cushion

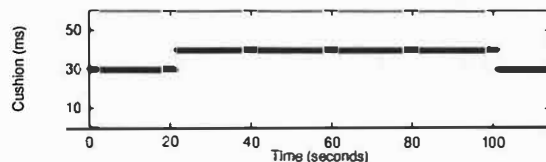


Figure 7: Stable cushion

Figure 6 shows the estimated cushion for the data presented in figure 4. A relatively small averaging period was used (200 measurements), and therefore the cushion varies continuously in an attempt to match the changing workstation performance. Figure 7 shows the estimated cushion using a larger averaging period (2000 measurements). The estimate is a lot more stable and represents the trend in workstation load. By increasing the averaging period, a reduction in average delay and a small increase in audio gap has resulted.

It is important to note that the algorithm design aims to smooth out short-lived scheduling anomalies, and cannot provide a solution if the workstation cannot (on average) keep up with the requirements of the real-time process. This can be detected if there is a constant trend in cushion increase. In this case, the audio tool should reduce the amount of processing performed (load adaption). This work is planned as part of the EPSRC RAT project [13].

### 5.3 Buffer adjustment

Maintaining a given amount of audio in the device driver buffer can easily be accomplished. The output buffer is initialised with a given number of samples (cushion size). On each cycle of the program, we calculate how many samples remain in the device driver buffer using the elapsed time given by the read length:

samples left in cushion = cushion size – read length

The cushion is refilled by writing out the required number of samples. An outline of the code that achieves this is given in figure 8.

```
/* Initialise cushion */
write(fd, mixed_audio_buffer, cushion_size);
for (;;)
{
    /* Read all there is (non blocking) */
    elapsed_time = read(fd, input_buffer, MAX_SIZE);
    /* Refill cushion */
    write(fd, mixed_audio_buffer, elapsed_time);

    /* Do all other processing */
}
```

Figure 8: Maintaining the cushion size

When the elapsed time exceeds the cushion because of a scheduling anomaly, then we shouldn't write back as much as we read, because this will effectively increase the cushion size. The cushion should be re-initialised by writing out its full size in this case. To maintain synchronisation between the input and playback operations the extra samples that were not written out are discarded. As these samples correspond to the silence period that resulted from the cushion overrun, no additional disturbance results from us discarding them and the timing relationship in the played out signal is maintained.

#### 5.3.1 Varying the cushion size

The methods described here vary the cushion size in line with the desired value. The cushion size reflects the current state of the workstation, which may change in response to other processing on the workstation.

Changing the cushion size during silence periods has minimal effect on the perception of audio [20]. It is

thus preferable to make any adjustments necessary during such periods.

However, it is possible to alter the cushion size during a talkspurt. A decrease in size will result in a reduction of the end-to-end delay that will be perceived at the end of the talkspurt. This can be achieved by simply writing out fewer samples than those needed to refill the current cushion. At this stage we have to decide what to do with the remaining samples that we did not transfer to the audio device. The simplest solution is to discard them. If the change in cushion size is small enough, as is usually the case, then the missing samples will not be noticed [20].

Increasing the cushion size is not as easy as decreasing it, since extra samples are needed. However, there may be extra audio available if the decision to increase the cushion follows a scheduling anomaly (see section 5.3). If there isn't extra audio available to fill the gap, then it has to be artificially created. Techniques for artificially creating extra audio required during periods of sample loss have been extensively studied by the authors [3, 5].

## 6 Discussion of results

The adaptive cushion algorithm has been implemented and tested in RAT. It has produced a perceivable reduction in the end-to-end delay of the system.

A simulator was built to evaluate the performance of the adaptive cushion algorithm. The simulator inputs load measurements and talkspurt information, and uses these to calculate the resulting delay and audio gaps for different adaptation strategies. In our experiments, we used real load measurements collected from RAT, and modelled talkspurt and pause durations after statistical data given in [21]. The data presented below uses load information collected on a medium loaded Sun Sparc 10 workstation running Solaris 2.4 over a period of 20 minutes. Results were collected during a multicast multimedia conference, while transmitting and receiving audio with RAT and moving video with vic [22].

| Adaptation | Read % | Avg del | Del $\sigma$ | Max del |
|------------|--------|---------|--------------|---------|
| No cushion |        | 43      | 36           | 240     |
| 195/200    | 97.5   | 31      | 4            | 70      |
| 970/1000   | 97     | 30      | 2            | 40      |
| 1800/2000  | 90     | 30      | 0            | 30      |
| 1970/2000  | 98.5   | 32      | 4            | 40      |

Table 1: Audio delay results (in ms)

Table 1 shows resulting end of talkspurt delay values for different adaptation conditions. The first row represents operation of the audio tool without the use of the cushion mechanism - as is used in all other existing audio

tools. The remaining rows represent results from the use of different parameters in the adaptive cushion algorithm. The two values in the column describing the adaptation parameters show the number of load measurements that are required to be covered by the cushion and the length of the history that is kept. The column labeled "Read %" gives the ratio of these two values indicating the percentage of measurements that are expected to be covered by the cushion. For each adaptation method the average, standard deviation and maximum of the delay are given in milliseconds.

Table 2 shows resulting audio gaps in talkspurts for the same adaptation conditions.

| Adaptation | Read % | Avg gap | Gap $\sigma$ | Total gap |
|------------|--------|---------|--------------|-----------|
| No cushion |        | 34      | 43           | 14340     |
| 195/200    | 97.5   | 21      | 47           | 8760      |
| 970/1000   | 97     | 21      | 48           | 9010      |
| 1800/2000  | 90     | 22      | 48           | 9090      |
| 1970/2000  | 98.5   | 20      | 46           | 8520      |

Table 2: Audio gap results (in ms)

It can be seen that the adaptive cushion algorithm results in improvement to both gap size and delay compared to the standard mechanism used in other audio tools.

## 7 Conclusion

The recent interest in multimedia conferencing is a result of the incorporation of cheap audio and video hardware in today's workstations, and also as a result of the development of a global infrastructure capable of supporting multimedia traffic - the Mbone. Audio quality is impaired by packet loss and variable delay in the network, and by lack of support for real-time applications in today's general purpose workstations.

This paper has presented an adaptive cushion algorithm that copes with the problems presented to real-time audio conferencing applications by a general purpose operating system. The continuity of the audio signal is ensured during scheduling anomalies by using the buffering capabilities of the audio device driver. The algorithm also restricts the amount of audio in the output audio device buffer to minimise the end-to-end delay. Negligible overhead in processing power is incurred with the adaptive cushion algorithm since a history of workstation load is built up incrementally over time.

The results presented in this paper show that there is a significant improvement in both minimisation of delay and audio gap size to be obtained from using the adaptive cushion algorithm.

## 8 Further work

Work is continuing in this area to tune the adaptive cushion algorithm. In particular we hope to identify a mapping between different user preferences and suitable history lengths for multi-way interactive multimedia conferences.

## 9 Acknowledgements

We would like to thank the multimedia group at UCL and in particular Jon Crowcroft, Mark Handley and Angela Sasse for their valuable input.

## References

- [1] Stephen Casner and Stephen Deering. First IETF Internet audiocast. *ACM Computer Communication Review*, 22(3):92-97, July 1992.
- [2] S. Deering. Host extensions for IP multicasting. Request for comments RFC 1112, Internet Engineering Task Force, August 1989. Obsoletes RFC 0988 1054.
- [3] Vicky Hardman, Martina Angela Sasse, Mark Handley, and Anna Watson. Reliable audio for use over the Internet. In *International Networking Conference (INET)*, 1995.
- [4] Lixia Zhang, Stephen Deering, Deborah Estrin, Scott Shenker, and Daniel Zappala. RSVP: a new resource ReSerVation protocol. *IEEE Network*, 7(5):8-18, September 1993.
- [5] V. Hardman, Kouvelas I., M.A. Sasse, and A. Watson. A packet loss Robust-Audio tool for use over the Mbone. Research Note RN/96/8, Dept. of Computer Science, University College London, England, 1996.
- [6] Newton Faller. Measuring the latency time of real-time Unix-like operating systems. Technical Report TR-92-037, International Computer Science Institute, Berkeley, California, June 1992.
- [7] Olof Hagsand and Peter Sjodin. Workstation support for real-time multimedia communication. In *Usenix Winter Technical Conference*, San Francisco, California, January 1994.
- [8] Sandeep Khanna, Michael Sebrée, and John Zolnowsky. Realtime scheduling in sunos 5.0. In *Usenix Winter Technical Conference*, 1992.

- [9] Sape J. Mullender, Ian M. Leslie, and Derek McAuley. Operating-system support for distributed multimedia. In *Usenix Summer Technical Conference*, Boston, Massachusetts, June 1994.
- [10] Tom Fisher. Real-time scheduling support in Ultrix-4.2 for multimedia communications. In *Third International Workshop on network and operating system support for digital audio and video*, pages 282–288, San Diego, California, November 1992. IEEE Communications Society.
- [11] M. Handley, P. Kirstein, and M. A. Sasse. Multimedia integrated conferencing for european researchers (MICE): piloting activities and the conference management and multiplexing centre. *Computer Networks and ISDN Systems*, 26(3):275–290, 1993.
- [12] J. Buckett, I. Cambell, T. J. Watson, M. A. Sasse, V. J. Hardman, and A. Watson. ReLaTe: Remote language teaching over superJANET. In *UKERNA 95 Networkshop*, University of Leicester, March 1995.
- [13] A.S. Sasse and Vicky Hardman. Multi-way multi-cast speech for multimedia conferencing over heterogeneous shared packet networks (rat - Robust-Audio tool). EPSRC project #GR/K72780, February 1996.
- [14] Van Jacobson and Steve McCanne. The LBL audio tool vat. Manual page, Available from <ftp://ftp.ee.lbl.gov/conferencing/vat/>, July 1992.
- [15] Henning Schulzrinne. Voice communication across the Internet: A network voice terminal. Technical Report TR 92-50, Dept. of Computer Science, University of Massachusetts, Amherst, Massachusetts, July 1992.
- [16] T. Turetti. The inria videoconferencing system (ivs). *ConneXions - The Interoperability Report*, 8(10):20–24, October 1994.
- [17] L. R. Rabiner and Schafer R. W. *Digital Processing of Speech Signals*. Prentice Hall, 1978.
- [18] Paul T. Brady. Effects of transmission delay on conversational behavior on echo-free telephone circuits. *Bell System Technical Journal*, 50:115–134, January 1971.
- [19] Van Jacobson. Multimedia conferencing on the Internet, August 1994. SIGCOMM '94 Tutorial.
- [20] John G. Gruber and Leo Strawczynski. Subjective effects of variable delay and speech clipping in dynamically managed voice systems. *IEEE Transactions on Communications*, 33(8):801–808, August 1985.
- [21] Paul T. Brady. A statistical analysis of on-off patterns in 16 conversations. *Bell System Technical Journal*, 47:73–91, January 1968.
- [22] Steve McCanne and Van Jacobson. vic: A flexible framework for packet video. In *Proc. of ACM Multimedia '95*, November 1995.

# On Designing Lightweight Threads for Substrate Software

Matthew Haines\*

Department of Computer Science  
University of Wyoming

`haines@cs.uwo.edu`

## Abstract

Existing user-level thread packages employ a “black box” design approach, where the implementation of the threads is hidden from the user. While this approach is often sufficient for application-level programmers, it hides critical *design decisions* that system-level programmers must be able to change in order to provide efficient service for high-level systems. By applying the principles of Open Implementation Analysis and Design, we construct a new user-level threads package that supports common thread abstractions and a well-defined meta-interface for altering the behavior of these abstractions. As a result, system-level programmers will have the advantages of using high-level thread abstractions without having to sacrifice performance, flexibility, or portability.

## 1 Introduction

Lightweight threads are useful for a variety of purposes. An application-level programmer will typically use threads to facilitate asynchronous scheduling for a number of related tasks. For example, consider an event-driven application, such as Xlib [25], where lightweight threads are used to schedule tasks for execution based on an external event. In this context, threads free the programmer from the details of dynamic scheduling. Fine-grain control over the behavior of a thread is typically not needed. There is no shortage of lightweight thread packages for application-level programmers, and a short list of such systems would likely include pthreads [22] (the POSIX interface for lightweight threads [16]), Solaris threads [26], fast-threads [2], and cthreads [23].

Lightweight threads are also useful for supporting independent tasks generated by parallel or concur-

rent programming languages. In this context, system-level programmers use lightweight threads as a major building-block of a *multithreaded runtime system*. For example, each of the following languages is supported by a multithreaded runtime system: C++ [12], Fortran M [12], Opus [15], Orca [7], PC++ [8], Sisal [13], Split-C [11], and SR [3]. However, *none* of these multithreaded runtime systems employs a single thread package for all platform implementations, and few use *any* of the lightweight thread packages listed in the preceding paragraph. While this may be surprising, there are several reasons why multithreaded runtime system designers shy away from standard lightweight thread packages, including:

1. *Lack of flexibility.* Existing thread packages are implemented as black boxes, so it is almost always impossible to change the detailed behavior of threads, mutexes, run lists, etc. However, most multithreaded runtime systems require explicit control over scheduling decisions and the interaction of threads with a communication substrate. For example, the Panda runtime system [7], which supports the Orca programming language [5], requires preemptive scheduling of threads with priorities, and the ability to turn preemption off and explicitly poll for incoming messages when there are no active threads. On the other hand, the runtime system that supports the SR programming language [3] assumes non-preemptive scheduling, in which the scheduler is free to select the next thread to be executed from a list of runnable threads. Both languages support communication between multiple processors, and this communication directly affects thread scheduling.
2. *Lack of performance.* Existing thread packages are geared towards supporting application-level programmers, who typically require threads to behave as normal Unix processes would. However supporting this behavior, including proper

\*Supported in part by the National Aeronautics and Space Administration under NASA Contract No. NASA-19480, while the author was in residence at ICASE, NASA Langley Research Center, and by the University of Wyoming Faculty Grant-in-aid and International Travel Grant Programs.

signal handling, adds a good deal of overhead to the thread operations. In contrast, most multithreaded runtime systems want bare-bones threads that are very fast, and to which they will add the complexities they require. The key here is that the runtime system designer, rather than the thread package designer, is in control of the tradeoffs between functionality and performance.

3. *Lack of portability.* Multithreaded runtime systems must execute on a wide variety of hardware and operating system platforms, which is extremely problematic for most lightweight thread packages.
4. *Lack of information.* Multithreaded runtime system designers often require tracing information for debugging and statistics information for tuning. Existing thread packages provide little or no support for obtaining this sort of information about the execution of the threads.

The central problem with using existing thread packages to support multithreaded runtime systems is that most existing thread packages are designed as “black-boxes,” providing only a very limited number of ways in which behavior can be modified. This is typically limited to altering the default size for a thread stack and choosing between a small, fixed-number of pre-implemented thread scheduling policies. This is almost always too restrictive for system-level programmers, and so most end up “rolling their own” thread packages.

Though “black box” abstractions are effective for constructing complex systems because they hide the details of an implementation, they don’t always work because “there are times when the implementation strategy for a module cannot be determined before knowing how the module will be used in a particular system” [18]. As we’ve seen, this is particularly true for multithreaded runtime systems employing lightweight threads, where many of the implementation decisions are to be made by the runtime system designer, not the thread package designer. Recently, the object-oriented research community has been addressing the issue of improved design methodologies for substrate (system-level) software [9, 18, 19, 20, 27]. From this research, a new design methodology for substrate software has emerged, called Open Implementation [18, 19]. The basic idea behind this new design methodology is to open the proverbial black box using a well-defined interface, called a *meta-interface*, that describes how the abstractions provided in the user-interface are to behave.

In this paper we provide an Open Implementation analysis and design of lightweight, user-level threads.

As a proof-of-concept, we have implemented this design to create a thread library called **OpenThreads**. The goal of this research is to produce a lightweight threads library that provides both a simple user-level interface and a robust meta-level interface for altering the behavior of the abstractions, so that a single threads package can be efficient, flexible, and portable. In addition, we extend the Open Implementation design methodology to address portability concerns by defining a system-level interface that clearly defines underlying dependencies.

The remainder of this paper is divided into five sections. Section 2 provides background on threads and Open Implementation analysis and design. Section 3 provides the Open Implementation analysis and design for OpenThreads, and discusses the three interfaces interfaces. Section 4 provides a discussion of our design relating to performance and open issues. Section 5 outlines related research projects, and we conclude in Section 6.

## 2 Background

In this section we provide background information for readers unfamiliar with either lightweight, user-level threads or Open Implementation Analysis and Design.

### 2.1 User-level Threads

User-level threads provide the ability for a programmer to create and control multiple, independent units of execution entirely outside of the operating system kernel (i.e., in user-space). The *state* of these threads is often minimal, consisting usually of an execution stack allocated in heap space and the set of CPU registers, and so these threads are often termed *lightweight*.

Since the OS kernel controls addressing and scheduling for the CPU, user-level threads must be multiplexed atop one or more kernel-level entities, such as Unix processes [4], Mach kernel threads [1], or a Sun Lightweight Processes (LWP) [24]. This “multiplexing” is commonly referred to as *scheduling* of the user-level threads. The kernel-entity (hereafter referred to as a “process”) also provides a common address space that is shared by all threads multiplexed onto that process, and synchronization primitives are provided to keep the memory consistent. It is also possible for threads to have some amount of *thread-specific data* by storing pointers to this data on each thread stack.

Scheduling policies for lightweight threads can be broadly classified either as *non-preemptive*, in which a thread executes until completion or until it decides to willingly yield the processor, or as *preemptive*, in which a thread can be interrupted at an arbitrary point during its execution so that some other thread may ex-

ecute. Orthogonal to the issue of preemption, thread scheduling can incorporate a wide range of capabilities, including priorities, hand-off scheduling, and arbitrary run list structures (such as trees). Clearly, there are many design choices to be made for thread scheduling, and many runtime system designers will want to switch between various policies depending on the state of the system. If a user-level thread package is not useful to a system-level programmer, lack of control over scheduling is commonly at the root of the cause.

## 2.2 Open Implementation Analysis and Design

In [18], Kiczales introduces a new approach for the design of substrate software called Open Implementation, and in this section we summarize these ideas. The reader is encouraged to examine [18] and [20] for more details on this design philosophy.

We have already stated that black-box abstractions do not always work because “there are times when the best implementation strategy for a module cannot be determined without knowing how the module will be used in a particular situation” [18]. So, what happens when a programmer using a black-box abstraction is confronted with conflict between how the abstraction is implemented and how the abstraction *should be* implemented? Since the current implementation is hidden within the internal portion of the black-box (as depicted in Figure 1), the programmer must “code around” the problem. This results in either *hematomas of duplication* or *coding between the lines*.

A hematoma of duplication occurs when a system-level programmer writes his own threads package, ensuring that his performance and flexibility demands are met. In addition to increasing the size and complexity of the resulting system, hematomas can result in convoluted code above the runtime system, where the black-box thread package may still be used.

Coding between the lines occurs when a programmer writes code in a particularly contorted way to get the desired performance or functionality. For example, consider a multithreaded runtime system designer who plans to create and destroy many threads. If the underlying thread package does not provide a way to cache the thread control blocks and thread stacks, the programmer might have to create server threads that never really dies so that resource management overheads are minimized.

These examples demonstrate that black-box designs often hide too much of the implementation for substrate software. While some of the implementation decisions are details that can be (and should be) hidden without problem, others are crucial to writing efficient

software and should be exposed in a controlled manner. These crucial design decisions are called *dilemmas*.

The Open Implementation design depicted in Figure 1 provides a mechanism for exposing dilemmas to the programmer so that these crucial design decisions can be made on a per-application basis. This mechanism is represented as a new interface, called the *meta interface*, that is presented to the application programmer for altering the behavior of the abstractions presented in the user interface. The meta interface provides a clean and controlled mechanism for customizing the implementation of substrate software and is the key to the Open Implementation design philosophy. Section 3.2.2 details the meta interface for OpenThreads.

## 3 Design

In this section we outline the design of OpenThreads. In Section 3.1 we itemize the issues are faced when designing a lightweight thread package, and in Section 3.2 we explain how these issues are exposed to the user in terms of *interfaces*.

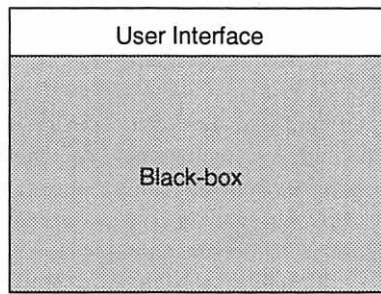
### 3.1 Dilemmas

We begin this section with an examination of the dilemmas that occur in the design of a thread package. As we stated in Section 2.2, the key difference between a black box design and an open implementation design is the level of control over these dilemmas.

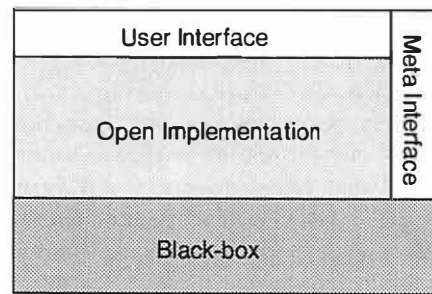
#### 3.1.1 Thread States

The lifetime of a thread is marked by a series of transformations between different *states*. A common set of thread states includes: being created, being placed onto an active run list, being selected (scheduled) for execution, being blocked on a mutex or condition variable, and being terminated. The transitions that take a thread from one state to the next can differ from one thread package to the next. For example, a blocked thread can either be resumed as an active thread or as a runnable thread. Figure 2 depicts these states transitions.

Besides the dilemma of where to place a thread that becomes unblocked, there are dilemmas associated with the transitions between the states. For example, what should be done when a thread is created or terminated? How about when a thread is in transition between the active and runnable states? In existing thread systems, these transitions are hidden from the user. This makes it impossible, for example, to trace the execution of a thread, since the user would need control over several transitions.



Traditional "Black-Box" Design



Open Implementation Design

Figure 1: "Black-box" and "Open Implementation" designs

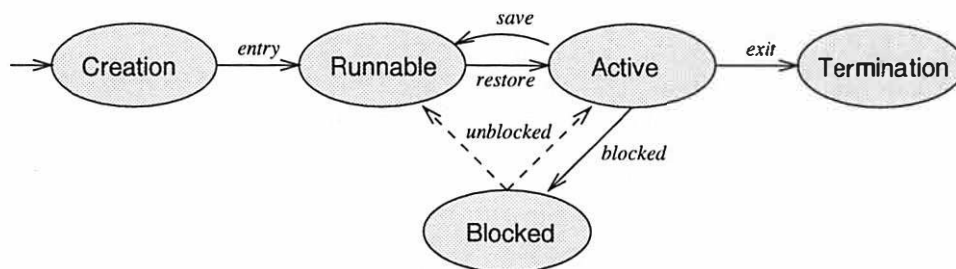


Figure 2: Thread states

Our approach to these dilemmas is to define a set of events that occur (perhaps repeatedly) during the life of a thread, and provide a mechanism for specifying user-defined actions to be performed whenever a thread encounters one of these events. As depicted in Figure 2, the following events are defined for a thread: **entry** - when a thread first begins execution; **exit** - when a thread is about to be terminated; **save** - when a thread moves from an active state to a runnable state; **restore**, when a thread moves from a runnable state to an active state; **blocked**, when a thread moves from an active state to a blocked state; and **unblocked**, when a thread moves from a blocked state to either an active or runnable state.

In addition to these thread-specific states, there are system states of either executing some runnable thread or being idle because all remaining threads are blocked. The idle system state is often achieved when all threads are waiting for some external event, such as a message or interrupt, to occur. The transition of the system from the active state to the idle state is also important, and so we define three more events to cover this situation: **idle begin** - when all threads in the system become blocked; **idle spin** - the duration of being idle; and **idle end** - when some thread becomes runnable again.

### 3.1.2 Thread Lists and Scheduling

A *thread list* is a data structure used to hold a collection of threads. The list may be used to hold threads that are ready to execute, called a *run list*, or may be used to hold threads in some other state, such as blocked on a mutex or condition variable. Since all threads must be on some thread list, state transitions typically involve moving a thread from one list to another. The scheduling policy for threads is therefore determined by the structure of the thread lists and the implementation of the operations that remove a thread from a list (**get**) and place a thread onto a list (**put**). For example, simple FIFO scheduling is achieved by using a FIFO queue for a run list, whereas priority scheduling might involve a tree of FIFO queues, where each leaf of the tree represents a queue of threads with the same priority.

Existing thread packages hide the concept of thread lists and provide abstract notions of scheduling for the system, such as FIFO or Round-Robin. However, this black-box approach prevents system-level programmers from gaining control over the most fundamental part of a thread package. For example, what if a tree structure would be most efficient for a run list, or what if multiple run lists are desired? What if mutex variables are to have thread lists which are scheduled differently from the run list, or from condition variable blocked lists? These dilemmas about the

nature and behavior of thread lists need to be exposed to the system-level programmer.

Our approach to these dilemmas is to make thread lists explicit, first-order entities in the system, and allow the user to define the `get` and `put` primitives for each list. Thread lists (or queues) are then explicitly associated with mutexes, condition variables, and runnable threads. OpenThreads provides mechanisms for specifying which list a thread is to be placed onto when yielding (`ot_thread_yield_onto`) and when initializing a thread (as an argument to `ot_thread_init`). Note that although there are multiple thread lists, only one at a time may be designated as the official “run list”, which is where ready threads are selected for execution. The run list is specified as an argument to the `ot_begin_mt` function.

### 3.1.3 Context Switching Modes

Every thread maintains *state information* that defines the thread. Minimally, this set includes the stack pointer, the instruction pointer, and the contents of the live registers. During a context switch between two threads, the state information of the old thread is saved and the state information of the new thread is restored.

Existing thread packages treat all context switches the same with respect to the state of a thread. However, this should not always be the case. Some threads, for example, may only use the integer registers, so saving and restoring the floating point registers at each context switch is a waste of precious cycles. But since the thread package designer doesn't know if threads will be using the floating point registers or not, a conservative decision is made and all registers are saved.

Our approach to these dilemmas is to allow for the context of a thread to be defined as either involving all registers, only the integer registers, or no registers. A more flexible approach might allow for a user-defined function that saves the exact thread state needed, but such a function would almost certainly be platform-dependent, thus violating portability.

### 3.1.4 Stack Management

Often, the most time-consuming portion of creating a new thread is allocating and aligning the thread stack. While most thread packages offer support to change the size of a thread's stack, few allow the programmer to specify the stack allocation policy. Lack of control over this dilemma is another common reason why multithreaded runtime systems abandon common thread packages. For example, the programmer may want to cache the stacks because threads are created and destroyed rapidly, but there are never more than a small number of threads alive at any one time. Another ex-

ample is when the programmer wishes to enable checks on stack overflow, or to resize the stack at runtime to enable stack growth.

Our approach to this dilemma is to allow the user to specify stack allocation and release policies.

### 3.1.5 Timing and Profiling

Existing thread systems offer little or no support for thread timing and profiling. While most application-level programmers may not care about how many times a given thread is switched or what the total execution time for a thread is, most system-level programmers do care about these measures. However, because the state transitions that define these measures are hidden from the user, it is usually impossible to gather these statistics even if the user wants to.

Our solution to this dilemma is to allow the user to take advantage of our exposed thread events and install monitoring code that will be executed whenever a thread reaches one of these events. For example, to count the number of times that a thread is switched from runnable to active, we can simply install the following function to be invoked whenever a thread triggers the *restore* event:

```
void bump () {  
    ctxswCounter++;  
}
```

## 3.2 Interfaces

We now discuss a mechanism for making these dilemmas available to the user in a clean and well-defined manner. Recall that in Figure 1, there are two interfaces presented to the user rather than just one. The first is the *user-level* interface, which defines the abstractions that are supported. The second is the *meta-level* interface, which defines how to change the behavior of the abstractions supported in the user-level interface. Thus, the meta-level interface represents the realization of our design dilemmas. We call this a *meta-interface* because it's an interface that describes how another interface (the user-interface) should behave.

### 3.2.1 The User-level Interface

The user-interface for OpenThreads provides a simple and clean mechanism for creating threads, mutex variables, condition variables, and thread lists. Noticeably absent are the plethora of routines that define the API for packages like pthreads, which is possible because the user-interface for OpenThreads is not concerned with modifying behavior. As with any thread package, OpenThreads allows the programmer to create new threads, yield, exit, wait for a mutex, and

```

extern void ot_init (int *argc, char *argv[], char *pkg_prefix);
extern void ot_start (ot_queue_t *runq);
extern void ot_end (void);

typedef void (ot_userf_t)(void *p0);
extern void ot_thread_init (ot_thread_t *thread, ot_userf_t *start,
                           void *args, unsigned tid, ot_queue_t *runq);
extern void ot_thread_yield (void);
extern void ot_thread_yield_onto (ot_queue_t *destq);
extern void ot_thread_exit (void);
extern unsigned ot_thread_id (void);
extern ot_thread_t *ot_current_thread (void);
extern void ot_thread_setspecific (ot_thread_t *thread, void *ptr,
                                   void (*cleanup)(void*));
extern void *ot_thread_getspecific (ot_thread_t *thread);

extern void ot_queue_init (ot_queue_t *q);

extern void ot_mutex_init (ot_mutex_t *m, ot_queue_t *blockq);
extern void ot_mutex_lock (ot_mutex_t *m);
extern int ot_mutex_trylock (ot_mutex_t *m);
extern void ot_mutex_unlock (ot_mutex_t *m);

extern void ot_cond_init (ot_cv_t *cv, ot_queue_t *blockq,
                        ot_mutex_t *m);
extern void ot_cond_wait (ot_cv_t *cv);
extern void ot_cond_bcast (ot_cv_t *cv);

```

Figure 3: OpenThreads user interface

block on a condition variable. These functions are detailed in Figure 3.

There are a few calls in this interface that bear special attention. First, the thread initialization function used to create a thread, `ot_thread_init`, takes a thread list (queue) as an argument, and places the newly created thread on that list. This gives the programmer explicit control over managing run lists. Second, the `ot_thread_yield_onto` routine is used to specify a destination thread list upon which the existing thread will be put. Again, this gives the programmer explicit control over thread list management. Third, the `ot_thread_setspecific` and `ot_thread_getspecific` calls are used to assign and retrieve a single generic pointer contained within the thread control block. This single pointer is meant to satisfy the needs of multithreaded runtime system designers, who all employ the concept of a *task* within their systems. Each task contains an instance of an OpenThread, which will perform the actual context switching between the tasks. However, since the current thread can only be defined in terms of an OpenThread, finding the current task requires a pointer from the OpenThread control block back to the surrounding task. Any additional thread-specific data can be multiplexed atop this single pointer without all users having to pay the cost in time and space to maintain an arbitrarily-long list of thread-specific pointers.

One problem with many thread packages is the vagueness with which multithreaded execution begins and ends, and what happens to the original thread of control within the process. OpenThreads makes these points of control explicit by creating two functions that mark the beginning and ending of multithreaded execution: `ot_begin_mt` and `ot_end_mt`, respectively. In between these calls the original process thread of control, now called the *process thread*, is allowed to execute any other code it desires, and is treated just like any other thread in the system. It can, for example, be blocked on a condition variable or be re-scheduled for execution on the run list. When the `ot_end_mt` call returns, the system is single-threaded again and another round of multithreaded execution may be initiated if desired. There are also functions for initializing the OpenThreads package (`ot_init`) and cleaning up after the package (`ot_done`). Sample code for a process initiating multithreaded execution is given in Figure 4.

### 3.2.2 The Meta-level Interface

The OpenThreads meta interface (Figure 5) provides the hooks needed to customize the design dilemmas listed in Section 3.1. In most cases, these decisions are set up as events that trigger user-specified actions, or *callback functions*, to occur. For exam-

ple, the `otm_push_callback` routine allows the user to specify a callback function to be invoked whenever the specified event is triggered. As mentioned in Section 3.1.1, events can be either *thread-specific*, which occurs whenever any thread enters a specific state, or *global*, which occurs when the system itself enters a new state. The valid thread-specific events are thread entry, thread exit, thread save, thread restore, thread blocking, and thread unblocking. The valid global events are system idle begin, system idle spin, and system idle end. Callback functions for each event are maintained in a stack, so that multiple functions can be associated with each event. For example, this would allow a set of tracing functions to be installed atop a set of timing functions. All function for an event are called in stack order, and the `otm_pop_callback` routine provides a way of clearing or rearranging the callback stack of a thread at any time.

The `otm_install_queue` function allows the user to provide implementations for the `get` and `put` functions of a thread list, as well as to define how large the link components of the thread list need to be. OpenThreads will invoke the `get` and `put` functions of a list whenever a scheduling decision needs to be made. This allows for multiple, user-defined thread lists to be used at the same time within OpenThreads, giving the user total control over scheduling. The interface allows different implementations to be associated with different thread lists at the same time.

The `otm_define.switch` routine allows the user to define the thread switching mode for a given thread (or for all threads). The valid switching modes are *all*, *integer*, or *none*, referring to the registers to be saved.

### 3.2.3 The System-level Interface

One of the key elements in the design of a multithreaded runtime system is *portability*. Since parallel and concurrent languages execute in a wide variety of environments, their runtime systems must support a wide-range of platforms.

To enable portability, we added a *system-level* interface (see Figure 6) to the traditional Open Implementation design, resulting in the overall design of OpenThreads with three interfaces, as depicted in Figure 7. The system interface provides a single place for mapping all dependencies that cannot be satisfied from within the OpenThreads implementation.

These routines are then mapped (usually with symbolic constants) onto platform-specific routines that provide the necessary functionality. Therefore, a successful port of OpenThreads requires modification of exactly one header file in a clean and well-defined way. Note that the routines at this level are decidedly low-level, so that additional overheads are not incurred. The thread initialization and context switch-

```

void main (int argc, char *argv[])
{
    ot_thread_t threads[NT];
    ot_queue_t runq;
    ...
    ot_init (&argc, argv, "ot-");
    ot_queue_init (&runq);
    for (int i = 0; i < NT; i++)
        ot_thread_init (&threads[i], entry_func, arg, &runq);
    ...
    ot_begin_mt (&runq);
    /* == begin multithreaded execution == */
    ...
    ot_end_mt ();
    /* == end multithreaded execution == */
    ...
    ot_done ();
}

```

Figure 4: Sample code for initiating multithreaded execution

```

extern void otm_init (void);
extern void otm_install_stackalloc (otm_sallocf_t *salloc,
    otm_sfreef_t *sfree);
extern void otm_define_switch (otm_switch_mode_t, ot_thread_t *t);
extern void otm_push_callback (int cbid, otm_callback_t *cbfunc);
extern otm_callback_t *otm_pop_callback (int cbid);
extern void otm_install_queue (ot_queue_t *q, unsigned qlink_size,
    unsigned qimp_size, otm_qinitf_t *init,
    otm_qgetf_t *get, otm_qputf_t *put);

```

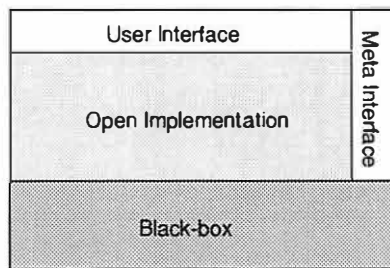
Figure 5: OpenThreads meta interface

```

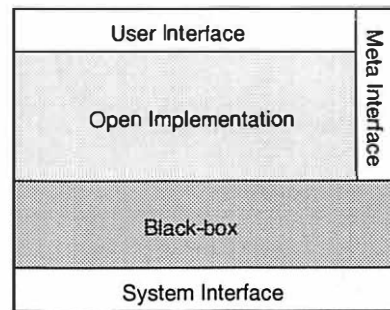
extern void *ots_stack_align (void *stack);
extern ots_stack_t *ots_stack_pointer (void *storage, int size);
extern ots_stack *ots_stack_init (ots_stack_t *sp, ots_userf_t *userf,
    void *userarg, ots_inif_t *initf, void *initarg);
extern void *ots_switch_all (ots_helperf_t *helper, void *arg1,
    void *arg2, ots_stack_t *new);
extern void *ots_lock_init (ots_lock_t lock);
extern int ots_lock_acquire (ots_lock_t lock);
extern void void ots_lock_release (ots_lock_t lock);

```

Figure 6: OpenThreads system interface



Traditional Open Implementation Design



OpenThreads Design

Figure 7: OpenThreads design

ing routines intentionally mimic the QuickThreads [17] macros, which are an excellent example of how very-low-level thread support can be provided in a machine-independent manner.

## 4 Discussion

In this section we discuss our design in terms of our original design goals and in terms of open issues that have yet to be addressed or resolved.

### 4.1 Design Goals

The design of OpenThreads is based on three essential goals: flexibility, efficiency, and portability. Our hypothesis is that substrate software requires success for each goal, and that existing lightweight thread packages fail in one or more of these goals. For example, one may argue that pthreads [16] is efficient and provides portability because its the “standard,” but it falls short in providing the flexibility demanded by most system-level programmers. For example, using pthreads it is impossible to trace thread execution when the scheduling policy is round-robin (preemptive).

We now examine our Open Implementation of lightweight threads with respect to these design goals:

#### 1. Flexibility.

The flexibility of our design is manifest by the meta-level interface, and the ability of a programmer to provide her own solutions to the design dilemmas outlined in Section 3.1. As a concrete example of its flexibility, we have adapted the Panda multithreaded runtime system [7] to use OpenThreads for implementing its tasks.

#### 2. Efficiency.

With regards to performance, we can report on the performance of OpenThreads in comparison

POSIX Pthreads and the QuickThreads package upon which OpenThreads is built. The comparison with Pthreads demonstrates the efficiency of OpenThreads for doing basic multithreading. The comparison with QuickThreads shows how much overhead we’ve added to the underlying low-level switching routines. As a demonstration of portability, we present these tests on three different processor architectures: the MIPS R4400, the Sun SPARC, and the DEC ALPHA. The measurements are given in Table 1. All numbers were gathered using averages for multiple runs, with an initial untimed run used to reduce cache miss effects. For context switches, OpenThreads saved all registers.

A second set of experiments evaluates the flexibility of OpenThreads to support the Panda runtime system. The hand-crafted Panda thread package was developed for the Amoeba operating distributed system and runs on 50 MHz SPARC processors with 32 Mbyte of memory and 4 Kbyte instruction and 2 Kbyte data caches (direct mapped). To achieve high performance the Panda scheduler is not built on top of Amoeba’s kernel threads, but it was derived from a user-level thread package developed at MIT by Wallach and Kaashoek. These results are presented in Table 2.

Given the extensive tuning performed on the Panda thread package and the overhead of OpenThreads, we expected the generic Panda threads implemented with OpenThreads (Panda-OT) to perform less than the hand-crafted Panda threads. The performance results presented in Table 2, however, show that Panda-OT has the fastest context switch time on Amoeba: 40.2  $\mu$ s (Panda) versus 35.4  $\mu$ s (Panda-OT). Examination of the low-level context switch code provided by QuickThreads (used in OpenThreads) revealed that it contains a SPARC specific opti-

|                  | MIPS R4400 |        | SPARC |        | ALPHA |        |
|------------------|------------|--------|-------|--------|-------|--------|
|                  | ctxsw      | create | ctxsw | create | ctxsw | create |
| QuickThreads STP | 0.9        | 5.6    | 6.5   | 10.5   | 1.2   | 2.7    |
| OpenThreads      | 1.3        | 7.0    | 7.2   | 13.5   | 2.3   | 3.1    |
| Pthreads         | 23.8       | 17.5   | 28.5  | 160.3  | —     | —      |

Table 1: OpenThreads raw performance on various architectures; *ctxsw* is the time in microseconds for a context switch and *create* is the time in microseconds to create a single thread with default stacksize (usually 8Kb).

|          | Amoeba |        | Solaris |        |
|----------|--------|--------|---------|--------|
|          | ctxsw  | create | ctxsw   | create |
| Panda    | 40.2   | 108.9  | 34.1    | 417.4  |
| Panda-OT | 35.4   | 262.7  | 7.4     | 62.0   |

Table 2: Performance of Panda thread packages on various platforms.

mization that saves one register-window underflow trap. Saving one trap to the Amoeba kernel accounts for approximately 7  $\mu$ s. Since we did not want to change Panda's implementation for the sake of making this comparison, we have provided the empirical numbers instead. However, the same optimization could be applied to the Panda threads, which would negate the performance advantage of OpenThreads for this experiment. In the end, we see about a 6% overhead for OpenThreads, which is a small price to pay for the added flexibility and portability.

Unlike the context switch results, the results for thread creation show that Panda-OT performs poorly in comparison to native Panda; creating a thread with Panda-OT is more than twice as expensive as with native Panda. This large overhead, however, is not a consequence of the flexibility of OpenThreads, but of a slight difference in scheduling policy of both thread packages. Panda-OT implements true priority scheduling, and therefore takes a scheduling decision as soon as the new thread is created and placed on the run queue. This results in a context switch to the newly created thread, which initializes itself and then terminates immediately. Once the thread has exited, another context switch occurs to transfer control back to the main thread. The Panda scheduler, on the other hand, does not take a scheduling decision until the main thread voluntarily yields control. The difference in scheduling policy causes Panda-OT to incur two additional context switches, thus accounting for the performance difference.

Table 2 also contains performance numbers for Panda-OT on Solaris. In this case we compare Panda-OT to the original Panda implementation that uses native Solaris threads. The performance was measured on a 70 Mhz SPARCstation 4 with 64 Mb of memory running Solaris 2.4. Panda-OT performs much better than native Panda: Panda-OT switches between threads over four times as fast and creates new threads over six times as fast. Since we do not have the source code of Solaris threads available, it is difficult to determine the exact causes of this great difference. We believe, however, that part of the explanation is Solaris ability to support a mixture of kernel and user threads. This functionality adds considerably to the complexity of the threads system, and requires expensive precautions to guard against preemption at various levels.

### 3. Portability.

Our system-level interface isolates all underlying dependencies in a single file that needs to be changed for each new platform. In some cases, there are no changes required at all. This is due to the fact that OpenThreads is currently mapping its low-level thread operations onto QuickThreads [17], which is already very portable. The Panda port runs the same code on both the Solaris and Amoeba platforms, and regular Orca programs were run to ensure the stability of the port.

OpenThreads has been tested on the Sun SuperSPARC and UltraSPARC architectures under both SunOS 4.1 and Solaris 2.5, the MIPS R4400 architecture under IRIX 5.3, and the Alpha AXP

architecture under DEC OSF-1. In addition, QuickThreads runs on the Intel 80x86, the Motorola 88000, the HP-PA, the KSR, and the VAX. As a result, OpenThreads can easily be ported to these architectures.

## 4.2 Open Issues

There are still a few open issues that remain in the design and implementation of OpenThreads.

1. Thread identification is the task of determining which thread, or more specifically, which thread control block is currently active at any given time. One way to do this is to reserve a global register to hold this pointer. This is the approach taken by Solaris threads [28], and has the advantage of being very fast and not requiring global memory. However, compiler and architecture support are required to reserve this register, and the loss of a register on RISC-based architectures is always cause for concern. Another approach is to keep the current thread pointer on a well-known offset in each thread stack [6]. This approach eliminates the need for both global memory and register space, but requires fancy stack alignments. Another approach, and the one currently employed by OpenThreads, is to use a global variable for storing the current thread pointer. This approach is the easiest to implement, does not require compiler support for extra registers, and does not require fancy stack manipulation. However, it does require per-processor global memory. A formal investigation regarding the best method for thread identification is still an open issue.
2. Kernel threads represent an opportunity to increase processor utilization in the face of blocking kernel calls, such as I/O. However, multiplexing user-level threads atop kernel threads requires a little finesse. We are in the process of revising OpenThreads to be safe in the presence of kernel threads. This requires protecting critical regions with kernel locks and identifying all global data as either shared among the kernel threads or private. Kernel thread private global data, such as the pointer to the current thread, needs to be stored as thread-specific data for each kernel thread. Powell et.al. [24] provide a discussion of this mapping for Solaris threads mapped onto LWPs. Another issue to be addressed in supporting kernel threads is the impact of kernel thread decisions on the meta-level interface. For example, some kernel threads might support features that allow for better optimization of the user threads, such as upcalls. To what degree

should these decisions be supported in the meta-interface? Kernel threads also raise the specter of multiprocessor issues, which we have completely ignored to this point.

3. Multithreaded runtime systems require both thread and communication components, and both must work well together. We are in the process of examining the issues regarding the combination of threads with communication models, and plan to test the flexibility of our system for performing platform-independent optimizations using a combination of thread and communication modules. Other systems combining threads with communication primitives include Chant [14], Nexus [12], PVM-threads [21], and MPI-threads [10].
4. Signals. Most papers on lightweight threads include long and involved discussions about signal handling in the presence of threads. We will avoid this discussion for now, and simply state that OpenThreads currently exposes all threads to the same signal mask. We do, however, ensure that if a signal arrives while the system is in an atomic section, the signal handler is delayed until after the atomic section has been completed.
5. Debugging. Debugging multithreaded systems has always been a trying experience because there are almost no debuggers that recognize the threads. OpenThreads does provide critical support in this regard by allowing traces to be made for thread state transitions by installing print statements at the various thread-specific event points. However, more sophisticated debugging tools are clearly required.

## 5 Related Research

OpenThreads represents a novel approach to the design of user-level threads, in which the user is given the opportunity to change the behavior of high-level abstractions in a well-defined manner. Many thread packages, such as pthreads [16], support an extensive user-interface with some behavior-modifying commands intertwined (such as attribute specification for threads). However, these systems do not take a systematic approach to exposing the critical design dilemmas and, as a result, fall short in providing the flexibility required by most system-level programmers.

QuickThreads [17] is a thread-building toolkit that offers platform independent micro-instructions for managing thread stacks. QuickThreads is similar to assembly language programming in terms of flexibility, speed, and complexity. OpenThreads builds on the QuickThreads design philosophy of keeping things

simple, and provides high-level abstractions whose behavior can be modified by the user in a well-defined manner.

The initial description of Open Implementation Analysis and Design [18] provided the motivation for much of this work. However, the initial description fails to talk about portability concerns. As a result, we extended the design to include a new system-level interface that unifies and defines all system dependencies.

## 6 Conclusions

It would seem that the last thing we need these days is another user-level thread package. From the standpoint of an application-level programmer I would have to agree. However, from the standpoint of a system-level programmer building multithreaded runtime systems, I would disagree. The evidence suggests that none of the current thread packages are being widely used by system-level programmers. In this paper we introduce the design of a user-level thread package for substrate software. The idea here is to identify all of the crucial design dilemmas that occur in building a thread package, and provide a clean and well-defined way for users to change these decisions. The result is a thread package with a simple user interface and a powerful meta interface for changing the behavior of the abstractions defined by the user interface. A system interface should also be used to isolate and define all underlying dependencies.

We have designed and built OpenThreads as a proof-of-concept for the ideas outlined in this paper, and are in the process of adapting several multithreaded runtime systems to use OpenThreads. We will report on the success of these attempts in the future.

## 7 Acknowledgments

Thanks to Koen Langendoen for helpful comments on a preliminary version of this paper, for his patience in describing the details of the Panda runtime system, and for helping me to debug OpenThreads in the process of porting Panda. It was a fun three days. Thanks also to Greg Benson for his description of the SR runtime system, and for his views on thread identification mechanisms. Finally, I thank Orran Krieger and the anonymous reviewers, whose detailed comments greatly helped to improve this paper.

## References

- [1] J. J. Accetta, R. V. Baron, W. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian, and M. W.

Young. Mach: A new kernel foundation for UNIX development, July 1986.

- [2] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *ACM Symposium on Operating Systems Principles*, pages 95–109, 1991.
- [3] Gregory R. Andrews and Ronald A. Olsson. *The SR Programming Language: Concurrency in Practice*. Benjamin/Cummings, 1993. ISBN 0-8053-0088-0.
- [4] Maurice J. Bach. *The Design of the UNIX Operating System*. Software Series. Prentice-Hall, 1986.
- [5] Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.
- [6] Greg Benson. Information on thread identification using stacks. Personal communication, March 1996.
- [7] Raoul Bhoedjang, Tim Rühl, Rutger Hofman, Koen Langendoen, Henri Bal, and Frans Kaashoek. Panda: A portable platform to support parallel programming languages. In *Symposium on Experiences with Distributed and Multiprocessor Systems IV*, pages 213–226, San Diego, CA, September 1993.
- [8] F. Bodin, P. Beckman, D. Gannon, S. Yang, S. Kesavan, and B. Mohr. Implementing a parallel C++ runtime system for scalable parallel systems. In *Supercomputing*, pages 588–597, Portland, OR, November 1993.
- [9] Martin D. Carroll and Margaret A. Ellis. *Designing and Coding Reusable C++*. Addison Wesley, 1995.
- [10] Aswini K. Chowdappa, Anthony Skjellum, and Nathan E. Doss. Thread-safe message passing with P4 and MPI. Technical Report TR-CS-941025, Computer Science Department and NSF Engineering Research Center, Mississippi State University, April 1994.
- [11] David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel programming in Split-C. In *Supercomputing*, pages 262–273, Portland, OR, November 1993.

- [12] Ian Foster, Carl Kesselman, Robert Olson, and Steven Tuecke. Nexus: An interoperability layer for parallel and distributed computer systems. Technical Report Version 1.3, Argonne National Labs, December 1993.
- [13] Matthew Haines and Wim Bohm. Task management, virtual shared memory, and multithreading in a distributed memory implementation of Sisal. In *Parallel Architectures and Languages*, pages 12–23. Springer-Verlag Lecture Notes in Computer Science, Vol 694, 1993.
- [14] Matthew Haines, David Cronk, and Piyush Mehrotra. On the design of Chant: A talking threads package. In *Proceedings of Supercomputing*, pages 350–359, Washington D.C., November 1994. ACM/IEEE.
- [15] Matthew Haines, Bryan Hess, Piyush Mehrotra, John Van Rosendale, and Hans Zima. Runtime support for data parallel tasks. In *Proceedings of The Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 432–439, McLean VA, February 1995. IEEE.
- [16] IEEE. *Standard for Threads Interface to POSIX*, 1996. P1003.1c.
- [17] David Keppel. Tools and techniques for building fast portable threads packages. Technical Report UWCSE 93-05-06, University of Washington, 1993.
- [18] Gregor Kiczales. Foil for the workshop on open implementation. In *Proceedings of the Workshop on Open Implementation*, 1994. Available at <http://www.xerox.com/PARC/sp1/eca/oi.html>.
- [19] Gregor Kiczales, Robert DeLine, Arthur Lee, and Chris Maeda. Open implementation analysis and design of substrate software. In *Tutorial Notes, OOPSLA 95*, Austin, TX, October 1995. ACM/SIGPLAN.
- [20] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.
- [21] Ravi Konuru, Jeremy Casas, Robert Prouty, Steve Otto, and Jonathan Walpole. A user-level process package for PVM. In *Proceedings of Scalable High Performance Computing Conference*, 1994.
- [22] Frank Mueller. A library implementation of POSIX threads under UNIX. In *Winter USENIX*, pages 29–41, San Diego, CA, January 1993.
- [23] Bodhisattwa Mukherjee. A portable and reconfigurable threads package. Technical Report GIT-ICS-91/02, College of Computing, Georgia Institute of Technology, Atlanta, Georgia, June 1991. Also appears in *Proceedings of Sun User's Group Technical Conference*, pages 101–112.
- [24] M.L. Powell, S.R. Kleinman, S. Barton, D. Shah, D. Stein, and M. Weeks. SunOS Multi-thread Architecture. In *Proceedings of USENIX Winter Technical Conference*, Dallas, TX, 1991.
- [25] Carl Schmidtman, Michael Tao, and Steven Watt. Design and implementation of a multithreaded Xlib. In *Winter USENIX*, pages 193–203, San Diego, CA, January 1993.
- [26] SunSoft Manual Set. *Solaris Multithreaded Programming Guide*. SunSoft Press, 1996. ISBN 0-13-160896-7.
- [27] W. Staringer. Constructing applications from reusable components. *IEEE Software*, 11(5):61–68, September 1994.
- [28] D. Stein and D. Shah. Implementing lightweight threads. In *Proceedings of USENIX Summer Technical Conference*, San Antonio, TX, July 1992.



# High-Performance Local Area Communication With Fast Sockets

Steven H. Rodrigues

Thomas E. Anderson

David E. Culler

*Computer Science Division  
University of California at Berkeley  
Berkeley, CA 94720*

## Abstract

Modern switched networks such as ATM and Myrinet enable low-latency, high-bandwidth communication. This performance has not been realized by current applications, because of the high processing overheads imposed by existing communications software. These overheads are usually not hidden with large packets; most network traffic is small. We have developed Fast Sockets, a local-area communication layer that utilizes a high-performance protocol and exports the Berkeley Sockets programming interface. Fast Sockets realizes round-trip transfer times of 60 microseconds and maximum transfer bandwidth of 33 MB/second between two UltraSPARC Is connected by a Myrinet network. Fast Sockets obtains performance by collapsing protocol layers, using simple buffer management strategies, and utilizing knowledge of packet destinations for direct transfer into user buffers. Using *receive posting*, we make the Sockets API a single-copy communications layer and enable regular Sockets programs to exploit the performance of modern networks. Fast Sockets transparently reverts to standard TCP/IP protocols for wide-area communication.

---

This work was supported in part by the Defense Advanced Research Projects Agency (N00600-93-C-2481, F30602-95-C-0014), the National Science Foundation (CDA 0401156), California MICRO, the AT&T Foundation, Digital Equipment Corporation, Exabyte, Hewlett Packard, Intel, IBM, Microsoft, Mitsubishi, Siemens Corporation, Sun Microsystems, and Xerox. Anderson and Culler were also supported by National Science Foundation Presidential Faculty Fellowships, and Rodrigues by a National Science Foundation Graduate Fellowship. The authors can be contacted at {steverod, tea, culler}@cs.berkeley.edu.

## 1 Introduction

The development and deployment of high performance local-area networks such as ATM [de Prycker 1993], Myrinet [Seitz 1994], and switched high-speed Ethernet has the potential to dramatically improve communication performance for network applications. These networks are capable of microsecond latencies and bandwidths of hundreds of megabits per second; their switched fabrics eliminate the contention seen on shared-bus networks such as traditional Ethernet. Unfortunately, this raw network capacity goes unused, due to current network communication software.

Most of these networks run the TCP/IP protocol suite [Postel 1981b, Postel 1981c, Postel 1981a]. TCP/IP is the default protocol suite for Internet traffic, and provides inter-operability among a wide variety of computing platforms and network technologies. The TCP protocol provides the abstraction of a reliable, ordered byte stream. The UDP protocol provides an unreliable, unordered datagram service. Many other application-level protocols (such as the FTP file transfer protocol, the Sun Network File System, and the X Window System) are built upon these two basic protocols.

TCP/IP's observed performance has not scaled to the ability of modern network hardware, however. While TCP is capable of sustained bandwidth close to the rated maximum of modern networks, actual bandwidth is very much implementation-dependent. The round-trip latency of commercial TCP implementations is hundreds of microseconds higher than the minimum possible on these networks. Implementations of the simpler UDP protocol, which lack the reliability and ordering mechanisms of TCP, perform little better [Keeton et al. 1995, Kay & Pasquale 1993, von Eicken et al. 1995]. This poor performance is due to the high per-packet processing costs (*processing overhead*) of the protocol implementations. In

local-area environments, where on-the-wire times are small, these processing costs dominate small-packet round-trip latencies.

Local-area traffic patterns exacerbate the problems posed by high processing costs. Most LAN traffic consists of small packets [Kleinrock & Naylor 1974, Schoch & Hupp 1980, Feldmeier 1986, Amer et al. 1987, Cheriton & Williamson 1987, Gusella 1990, Caceres et al. 1991, Claffy et al. 1992]. Small packets are the rule even in applications considered bandwidth-intensive: 95% of all packets in a NFS trace performed at the Berkeley Computer Science Division carried less than 192 bytes of user data [Dahlin et al. 1994]; the mean packet size in the trace was 382 bytes. Processing overhead is the dominant transport cost for packets this small, limiting NFS performance on a high-bandwidth network. This is true for other applications: most application-level protocols in local-area use today (X11, NFS, FTP, etc.) operate in a *request-response*, or *client-server*, manner: a client machine sends a small request message to a server, and awaits a response from the server. In the request-response model, processing overhead usually cannot be hidden through packet pipelining or through overlapping communication and computation, making round-trip latency a critical factor in protocol performance.

Traditionally, there are several methods of attacking the processing overhead problem: changing the application programming interface (API), changing the underlying network protocol, changing the implementation of the protocol, or some combination of these approaches. *Changing the API* modifies the code used by applications to access communications functionality. While this approach may yield better performance for new applications, legacy applications must be re-implemented to gain any benefit. *Changing the communications protocol* changes the “on-the-wire” format of data and the actions taken during a communications exchange — for example, modifying the TCP packet format. A new or modified protocol may improve communications performance, but at the price of incompatibility: applications communicating via the new protocol are unable to share data directly with applications using the old protocol. *Changing the protocol implementation* rewrites the software that implements a particular protocol; packet formats and protocol actions do not change, but the code that performs these actions does. While this approach provides full compatibility with existing protocols, fundamental limitations of the protocol design may limit the performance gain.

Recent systems, such as Active Messages [von Eicken et al. 1992], Remote Queues [Brewer et al.

1995], and native U-Net [von Eicken et al. 1995], have used the first two methods; they implement new protocols and new programming interfaces to obtain improved local-area network performance. The protocols and interfaces are lightweight and provide programming abstractions that are similar to the underlying hardware. All of these systems realize latencies and throughput close to the physical limits of the network. However, none of them offer compatibility with existing applications.

Other work has tried to improve performance by re-implementing TCP. Recent work includes zero-copy TCP for Solaris [Chu 1996] and a TCP interface for the U-Net interface [von Eicken et al. 1995]. These implementations can inter-operate with other TCP/IP implementations and improve throughput and latency relative to standard TCP/IP stacks. Both implementations can realize the full bandwidth of the network for large packets. However, both systems have round-trip latencies considerably higher than the raw network.

This paper presents our solution to the overhead problem: a new communications protocol and implementation for local-area networks that exports the Berkeley Sockets API, uses a low-overhead protocol for local-area use, and reverts to standard protocols for wide-area communication. The Sockets API is a widely used programming interface that treats network connections as files; application programs read and write network connections exactly as they read and write files. The Fast Sockets protocol has been designed and implemented to obtain a low-overhead data transmission/reception path. Should a Fast Sockets program attempt to connect with a program outside the local-area network, or to a non-Fast Sockets program, the software transparently reverts to standard TCP/IP sockets. These features enable high-performance communication through relinking existing application programs.

Fast Sockets achieves its performance through a number of strategies. It uses a lightweight protocol and efficient buffer management to minimize book-keeping costs. The communication protocol and its programming interface are integrated to eliminate module-crossing costs. Fast Sockets eliminates copies within the protocol stack by using knowledge of packet memory destinations. Additionally, Fast Sockets was implemented without modifications to the operating system kernel.

A major portion of Fast Sockets’ performance is due to *receive posting*, a technique of utilizing information from the API about packet destinations to minimize copies. This paper describes the use of receive posting in a high-performance communications stack. It also describes the design and implementation of a

low-overhead protocol for local-area communication.

The rest of the paper is organized as follows. Section 2 describes problems of current TCP/IP and Sockets implementations and how these problems affect communication performance. Section 3 describes how the Fast Sockets design attempts to overcome these problems. Section 4 describes the performance of the resultant system, and section 5 compares Fast Sockets to other attempts to improve communication performance. Section 6 presents our conclusions and directions for future work in this area.

## 2 Problems with TCP/IP

While TCP/IP can achieve good throughput on currently deployed networks, its round-trip latency is usually poor. Further, observed bandwidth and round-trip latencies on next-generation network technologies such as Myrinet and ATM do not begin to approach the raw capabilities of these networks [Keeton et al. 1995]. In this section, we describe a number of features and problems of commercial TCP implementations, and how these features affect communication performance.

### 2.1 Built for the Wide Area

TCP/IP was originally designed, and is usually implemented, for wide-area networks. While TCP/IP is usable on a local-area network, it is not optimized for this domain. For example, TCP uses an in-packet checksum for end-to-end reliability, despite the presence of per-packet CRC's in most modern network hardware. But computing this checksum is expensive, creating a bottleneck in packet processing. IP uses header fields such as 'Time-To-Live' which are only relevant in a wide-area environment. IP also supports internetwork routing and in-flight packet fragmentation and reassembly, features which are not useful in a local-area environment. The TCP/IP model assumes communication between autonomous machines that cooperate only minimally. However, machines on a local-area network frequently share a common administrative service, a common file system, and a common user base. It should be possible to extend this commonality and cooperation into the network communication software.

### 2.2 Multiple Layers

Standard implementations of the Sockets interface and the TCP/IP protocol suite separate the protocol and interface stack into multiple layers. The Sockets

interface is usually the topmost layer, sitting above the protocol. The protocol layer may contain sub-layers: for example, the TCP protocol code sits above the IP protocol code. Below the protocol layer is the interface layer, which communicates with the network hardware. The interface layer usually has two portions, the network programming interface, which prepares outgoing data packets, and the network device driver, which transfers data to and from the network interface card (NIC).

This multi-layer organization enables protocol stacks to be built from many combinations of protocols, programming interfaces, and network devices, but this flexibility comes at the price of performance. Layer transitions can be costly in time and programming effort. Each layer may use a different abstraction for data storage and transfer, requiring data transformation at every layer boundary. Layering also restricts information transfer. Hidden implementation details of each layer can cause large, unforeseen impacts on performance [Clark 1982, Crowcroft et al. 1992]. Mechanisms have been proposed to overcome these difficulties [Clark & Tennenhouse 1990], but existing work has focused on message throughput, rather than protocol latency [Abbott & Peterson 1993]. Also, the number of programming interfaces and protocols is small: there are two programming interfaces (Berkeley Sockets and the System V Transport Layer Interface) and only a few data transfer protocols (TCP/IP and UDP/IP) in widespread usage. This paucity of distinct layer combinations means that the generality of the multi-layer organization is wasted. Reducing the number of layers traversed in the communications stack should reduce or eliminate these layering costs for the common case of data transfer.

### 2.3 Complicated Memory Management

Current TCP/IP implementations use a complicated memory management mechanism. This system exists for a number of reasons. First, a multi-layered protocol stack means packet headers are added (or removed) as the packet moves downward (or upward) through the stack. This should be done easily and efficiently, without excessive copying. Second, buffer memory inside the operating system kernel is a scarce resource; it must be managed in a space-efficient fashion. This is especially true for older systems with limited physical memory.

To meet these two requirements, mechanisms such as the Berkeley Unix `mbuf` have been used. An `mbuf` can directly hold a small amount of data, and `mbuf`s can be chained to manage larger data sets. Chain-

ing makes adding and removing packet headers easy. The `mbuf` abstraction is not cheap, however: 15% of the processing time for small TCP packets is consumed by `mbuf` management [Kay & Pasquale 1993]. Additionally, to take advantage of the `mbuf` abstraction, user data must be copied into and out of `mbufs`, which consumes even more time in the data transfer critical path. This copying means that nearly one-quarter of the small-packet processing time in a commercial TCP/IP stack is spent on memory management issues. Reducing the overhead of memory management is therefore critical to improving communications performance.

### 3 Fast Sockets Design

Fast Sockets is an implementation of the Sockets API that provides high-performance communication and inter-operability with existing programs. It yields high-performance communication through a low-overhead protocol layered on top of a low-overhead transport mechanism (Active Messages). Interoperability with existing programs is obtained by supporting most of the Sockets API and transparently using existing protocols for communication with non-Fast Sockets programs. In this section, we describe the design decisions and consequent trade-offs of Fast Sockets.

#### 3.1 Built For The Local Area

Fast Sockets is targeted at local-area networks of workstations, where processing overhead is the primary limitation on communications performance. For low-overhead access to the network with a defined, portable interface, Fast Sockets uses Active Messages [von Eicken et al. 1992, Martin 1994, Culler et al. 1994, Mainwaring & Culler 1995]. An *active message* is a network packet which contains the name of a *handler function* and data for that handler. When an active message arrives at its destination, the handler is looked up and invoked with the data carried in the message. While conceptually similar to a remote procedure call [Birrell & Nelson 1984], an active message is constrained in the amount and types of data that can be carried and passed to handler functions. These constraints enable the structuring of an Active Messages layer for high performance. Also, Active Messages uses protected user-level access to the network interface, removing the operating system kernel from the critical path. Active messages are reliable, but not ordered.

Using Active Messages as a network transport in-

volves a number of trade-offs. Active Messages has its own 'on-the-wire' packet format; this makes a full implementation of TCP/IP on Active Messages infeasible, as the transmitted packets will not be comprehensible by other TCP/IP stacks. Instead, we elected to implement our own protocol for local area communication, and fall back to normal TCP/IP for wide-area communication. Active Messages operates primarily at user-level; although access to the network device is granted and revoked by the operating system kernel, data transmission and reception is performed by user-level code. For maximum performance, Fast Sockets is written as a user-level library. While this organization avoids user-kernel transitions on communications events (data transmission and reception), it makes the maintenance of shared and global state, such as the TCP and UDP port name spaces, difficult. Some global state can be maintained by simply using existing facilities. For example, the port name spaces can use the in-kernel name management functions. Other shared or global state can be maintained by using a server process to store this state, as described in [Maeda & Bershad 1993b]. Finally, using Active Messages limits Fast Sockets communication to the local-area domain. Fast Sockets supports wide-area communication by automatically switching to standard network protocols for non-local addresses. It is a reasonable trade-off, as endpoint processing overheads are generally not the limiting factor for internet-work communication.

Active Messages does have a number of benefits, however. The handler abstraction is extremely useful. A handler executes upon message reception at the destination, analogous to a network interrupt. At this time, the protocol can store packet data for later use, pass the packet data to the user program, or deal with exceptional conditions. Message handlers allow for a wide variety of control operations within a protocol without slowing down the critical path of data transmission and reception. Handler arguments enable the easy separation of packet data and metadata: packet data (that is, application data) is carried as a bulk transfer argument, and packet metadata (protocol headers) are carried in the remaining word-sized arguments. This is only possible if the headers can fit into the number of argument words provided; for our local-area protocol, the 8 words supplied by Active Messages is sufficient.

Fast Sockets further optimizes for the local area by omitting features of TCP/IP unnecessary in that environment. For example, Fast Sockets uses the checksum or CRC of the network hardware instead of one in the packet header; software checksums make little sense when packets are only traversing a single net-

work. Fast Sockets has no equivalent of IP's 'Time-To-Live' field, or IP's internetwork routing support. Since maximum packet sizes will not change within a local area network, Fast Sockets does not support IP-style in-flight packet fragmentation.

### 3.2 Collapsing Layers

To avoid the performance and structuring problems created by the multi-layered implementation of Unix TCP/IP, Fast Sockets collapses the API and protocol layers of the communications stack together. This avoids abstraction conflicts between the programming interface and the protocol and reduces the number of conversions between layer abstractions, further reducing processing overheads.

The actual network device interface, Active Messages, remains a distinct layer from Fast Sockets. This facilitates the portability of Fast Sockets between different operating systems and network hardware. Active Messages implementations are available for the Intel Paragon [Liu & Culler 1995], FDDI [Martin 1994], Myrinet [Mainwaring & Culler 1995], and ATM [von Eicken et al. 1995]. Layering costs are kept low because Active Messages is a thin layer, and all of its implementation-dependent constants (such as maximum packet size) are exposed to higher layers.

The Fast Sockets layer stays lightweight by exploiting Active Message handlers. Handlers allow rarely-used functionality, such as connection establishment, to be implemented without affecting the critical path of data transmission and reception. There is no need to test for uncommon events when a packet arrives — this is encoded directly in the handler. Unusual data transmission events, such as *out-of-band* data, also use their own handlers to keep normal transfer costs low.

Reducing the number of layers and exploiting Active Message handlers lowers the protocol- and API-specific costs of communication. While collapsing layers means that every protocol layer-API combination has to be written anew, the number of such combinations is relatively few, and the number of distinct operations required for each API is small.

### 3.3 Simple Buffer Management

Fast Sockets avoids the complexities of mbuf-style memory management by using a single, contiguous virtual memory buffer for each socket. Data is transferred directly into this buffer via Active Message data transfer messages. The message handler places data sequentially into the buffer to maintain in-order

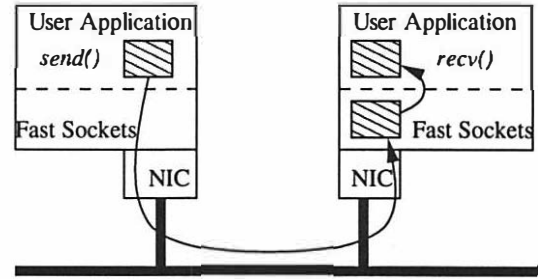


Figure 1: Data transfer in Fast Sockets. A `send()` call transmits the data directly from the user buffer into the network. When it arrives at the remote destination, the message handler places it into the socket buffer, and a subsequent `recv()` call copies it into the user buffer.

delivery and make data transfer to a user buffer a simple memory copy. The argument words of the data transfer messages carry packet metadata; because the argument words are passed separately to the handler, there is no need for the memory management system to strip off packet headers.

Fast Sockets eliminates send buffering. Because many user applications rely heavily on small packets and on request-response behavior, delaying packet transmission only serves to increase user-visible latency. Eliminating send-side buffering reduces protocol overhead because there are no copies on the send side of the protocol path — Active Messages already provides reliability.

Figure 1 shows Fast Sockets' send mechanism and buffering techniques.

A possible problem with this approach is that having many Fast Sockets consumes considerably more memory than the global mbuf pool used in traditional kernel implementations. This is not a major concern, for two reasons. First, the memory capacity of current workstations is very large; the scarce physical memory situations that the traditional mechanisms are designed for is generally not a problem. Second, the socket buffers are located in pageable virtual memory — if memory and scheduling pressures are severe enough, the buffer can be paged out. Although paging out the buffer will lead to worse performance, we expect that this is an extremely rare occurrence.

A more serious problem with placing socket buffers in user virtual memory is that it becomes extremely difficult to share the socket buffer between processes. Such sharing can arise due to a `fork()` call, for instance. Currently, Fast Sockets cannot be shared between processes (see section 3.7).

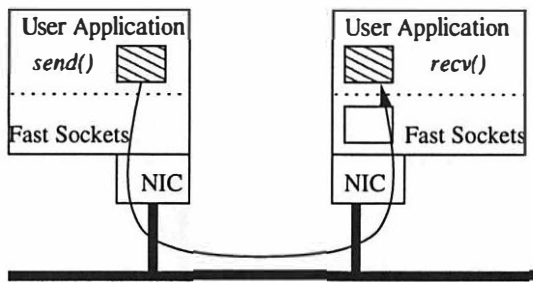


Figure 2: Data transfer via receive posting. If a `recv()` call is issued prior to the arrival of the desired data, the message handler can directly route the data into the user buffer, bypassing the socket buffer.

### 3.4 Copy Avoidance

The standard Fast Sockets data transfer mechanism involves two copies along the receive path, from the network interface to the socket buffer, and then from the socket buffer to the user buffer specified in a `recv()` call. While the copy from the network interface cannot be avoided, the second copy increases the processing overhead of a data packet, and consequently, round-trip latencies. A high-performance communications layer should bypass this second copy whenever possible.

It is possible, under certain circumstances, to avoid the copy through the socket buffer. If the data's final memory destination is already known upon packet arrival (through a `recv()` call), the data can be directly copied there. We call this technique *receive posting*. Figure 2 shows how receive posting operates in Fast Sockets. If the message handler determines that an incoming packet will satisfy an outstanding `recv()` call, the packet's contents are received directly into the user buffer. The socket buffer is never touched.

Receive posting in Fast Sockets is possible because of the integration of the API and the protocol, and the handler facilities of Active Messages. Protocol-API integration allows knowledge of the user's destination buffer to be passed down to the packet processing code. Using Active Message handlers means that the Fast Sockets code can decide where to place the incoming data when it arrives.

### 3.5 Design Issues In Receive Posting

Many high-performance communications systems are now using *sender-based memory management*, where the sending node determines the data's final memory destination. Examples of communications layers with this memory management style are Hamlyn

[Buzzard et al. 1996] and the SHRIMP network interface [Blumrich et al. 1994]. Also, the initial version of Generic Active Messages [Culler et al. 1994] offered only sender-based memory management for data transfer.

Sender-based memory management has a major drawback for use in byte-stream and message-passing APIs such as Sockets. With sender-based management, the sending and receiving endpoints must synchronize and agree on the destination of each packet. This synchronization usually takes the form of a message exchange, which imposes a time cost and a resource cost (the message exchange uses network bandwidth). To minimize this synchronization cost, the original version of Fast Sockets used the socket buffer as a default destination. This meant that when a `recv()` call was made, data already in flight had to be directed through the socket buffer, as shown in Figure 1. These synchronization costs lowered Fast Sockets' throughput relative to Active Messages' on systems where the network was not a limiting factor, such as the Myrinet network.

Generic Active Messages 1.5 introduced a new data transfer message type that did not require sender-based memory management. This message type, the *medium message*, transferred data into an anonymous region of user memory and then invoked the handler with a pointer to the packet data. The handler was responsible for long-term data storage, as the buffer was deallocated upon handler completion. Keeping memory management on the receiver improves Fast Sockets performance considerably. Synchronization is now only required between the API and the protocol layers, which is simple due to their integration. The receive handler now determines the memory destination of incoming data at packet arrival time, enabling a `recv()` of in-flight data to benefit from receive posting and bypass the socket buffer. The net result is that receiver-based memory management did not significantly affect Fast Sockets' round-trip latencies and improved large-packet throughput substantially, to within 10% of the throughput of raw Active Messages.

The use of receiver-based memory management has some trade-offs relative to sender-based systems. In a true zero-copy transport layer, where packet data is transferred via DMA to user memory, transfer to an anonymous region can place an extra copy into the receive path. This is not a problem for two reasons. First, many current I/O architectures, like that on the SPARC, are limited in the memory regions that they can perform DMA operations to. Second, DMA operations usually require memory pages to be pinned in physical memory, and pinning an arbitrary page can

be an expensive operation. For these reasons, current versions of Active Messages move data from the network interface into an anonymous staging area before either invoking the message handler (for medium messages) or placing the data in its final destination (for standard data transfer messages). Consequently, receiver-based memory management does not impose a large cost in our current system. For a true zero-copy system, it should be possible for a handler to be responsible for moving data from the network interface card to user memory.

Based on our experiences implementing Fast Sockets with both sender- and receiver-based memory management schemes, we believe that, for messaging layers such as Sockets, the performance increase delivered by receiver-based management schemes outweigh the implementation costs.

### 3.6 Other Fast Socket Operations

Fast Sockets supports the full sockets API, including socket creation, name management, and connection establishment. The `socket` call for the `AF_INET` address family and the “default protocol” creates a Fast Socket; this allows programs to explicitly request TCP or UDP sockets in a standard way. Fast Sockets utilizes existing name management facilities. Every Fast Socket has a *shadow socket* associated with it; this shadow socket is of the same type and shares the same file descriptor as the Fast Socket. Whenever a Fast Socket requests to `bind()` to an `AF_INET` address, the operation is first performed on the shadow socket to determine if the operation is legal and the name is available. If the shadow socket `bind` succeeds, the `AF_INET` name is bound to the Fast Socket’s Active Message endpoint name via an external name service. Other operations, such as `setsockopt()` and `getsockopt()`, work in a similar fashion.

Shadow sockets are also used for connection establishment. When a `connect()` call is made by the user application, Fast Sockets determines if the name is in the local subnet. For local addresses, the shadow socket performs a `connect()` to a port number that is determined through a hash function. If this `connect()` succeeds, then a handshake is performed to bootstrap the connection. Should the `connect()` to the hashed port number fail, or the connection bootstrap process fail, then a normal `connect()` call is performed. This mechanism allows a Fast Sockets program to connect to a non-Fast Sockets program without difficulties.

An `accept()` call becomes more complicated as a result of this scheme, however. Because both Fast

Sockets and non-Fast Sockets programs can connect to a socket that has performed a `listen()` call, there are two distinct port numbers for a given socket. The port supplied by the user accepts connection requests from programs using normal protocols. The second port is derived from hashing on the user port number, and is used to accept connection requests from Fast Sockets programs. An `accept()` call multiplexes connection requests from both ports.

The connection establishment mechanism has some trade-offs. It utilizes existing name and connection management facilities, minimizing the amount of code in Fast Sockets. Using TCP/IP to bootstrap the connection can impose a high time cost, which limits the realizable throughput of short-lived connections. Using two port numbers also introduces the potential problem of conflicts: the Fast Sockets-generated port number could conflict with a user port. We do not expect this to realistically be a problem.

### 3.7 Fast Sockets Limitations

Fast Sockets is a user-level library. This limits its full compatibility with the Sockets abstraction. First, applications must be relinked to use Fast Sockets, although no code changes are required. More seriously, Fast Sockets cannot currently be shared between two processes (for example, via a `fork()` call), and all Fast Sockets state is lost upon an `exec()` or `exit()` call. This poses problems for traditional Internet server daemons and for “super-server” daemons such as `inetd`, which depend on a `fork()` for each incoming request. User-level operation also causes problems for socket termination; standard TCP/IP sockets are gracefully shut down on process termination. These problems are not insurmountable. Sharing Fast Sockets requires an Active Messages layer that allows endpoint sharing and either the use of a dedicated server process [Maeda & Bershad 1993b] or the use of shared memory for every Fast Socket’s state. Recovering Fast Sockets state lost during an `exec()` call can be done via a dedicated server process, where the Fast Socket’s state is migrated to and from the server before and after the `exec()` — similar to the method used by the user-level Transport Layer Interface [Stevens 1990].

The Fast Sockets library is currently single-threaded. This is problematic for current versions of Active Messages because an application must explicitly touch the network to receive messages. Since a user application could engage in an arbitrarily long computation, it is difficult to implement operations such as asynchronous I/O. While multi-threading offers one solution, it makes the library less portable,

and imposes synchronization costs.

## 4 Performance

This section presents our performance measurements of Fast Sockets, both for microbenchmarks and a few applications. The microbenchmarks assess the success of our efforts to minimize overhead and round-trip latency. We report round-trip times and sustained bandwidth available from Fast Sockets, using both our own microbenchmarks and two publicly available benchmark programs. The true test of Fast Sockets' usefulness, however, is how well its raw performance is exposed to applications. We present results from an FTP application to demonstrate the usefulness of Fast Sockets in a real-world environment.

### 4.1 Experimental Setup

Fast Sockets has been implemented using Generic Active Messages (GAM) [Culler et al. 1994] on both HP/UX 9.0.x and Solaris 2.5. The HP/UX platform consists of two 99Mhz HP735's interconnected by an FDDI network and using the Medusa network adapter [Banks & Prudence 1993]. The Solaris platform is a collection of UltraSPARC 1's connected via a Myrinet network [Seitz 1994]. For all tests, there was no other load on the network links or switches.

Our microbenchmarks were run against a variety of TCP/IP setups. The standard HP/UX TCP/IP stack is well-tuned, but there is also a single-copy stack designed for use on the Medusa network interface. We ran our tests on both stacks. While the Solaris TCP/IP stack has reasonably good performance, the Myrinet TCP/IP drivers do not. Consequently, we also ran our microbenchmarks on a 100-Mbit Ethernet, which has an extremely well-tuned driver.

We used Generic Active Messages 1.0 on the HP/UX platform as a base for Fast Sockets and as our Active Messages layer. The Solaris tests used Generic Active Messages 1.5, which adds support for medium messages (receiver-based memory management) and client-server program images.

### 4.2 Microbenchmarks

#### 4.2.1 Round-Trip Latency

Our round-trip microbenchmark is a simple ping-pong test between two machines for a given transfer size. The ping-pong is repeated until a 95% confidence interval is obtained. TCP/IP and Fast Sockets use the same program, Active Messages uses different code but the same algorithm. We used the

| Layer                  | Per-Byte<br>( $\mu$ sec) | $t_0$<br>( $\mu$ sec) | Actual<br>Startup |
|------------------------|--------------------------|-----------------------|-------------------|
| Fast Sockets           | 0.068                    | 157.8                 | 57.8              |
| Active Messages        | 0.129                    | 38.9                  | 45.0              |
| TCP/IP (Myrinet)       | 0.076                    | 736.4                 | 533.0             |
| TCP/IP (Fast Ethernet) | 0.174                    | 366.2                 | 326.0             |
| Small Packets          |                          |                       |                   |
| Fast Sockets           | 0.124                    | 61.4                  | 57.8              |
| Active Messages        | 0.123                    | 45.4                  | 45.0              |
| TCP/IP (Myrinet)       | 0.223                    | 533.0                 | 533.0             |
| TCP/IP (Fast Ethernet) | 0.242                    | 325.0                 | 326.0             |

Table 1: Least-squares analysis of the Solaris round-trip microbenchmark. The per-byte and estimated startup ( $t_0$ ) costs are for round-trip latency, and are measured in microseconds. The actual startup costs (for a single-byte message) are also shown. Actual and estimated costs differ because round-trip latency is not strictly linear. Per-byte costs for Fast Sockets are lower than for Active Messages because Fast Sockets benefits from packet pipelining; the Active Messages test only sends a single packet at a time. "Small Packets" examines protocol behavior for packets smaller than 1K; here, Fast Sockets and Active Messages do considerably better than TCP/IP.

TCP/IP TCP\_NODELAY option to force packets to be transmitted as soon as possible (instead of the default behavior, which attempts to batch small packets together); this reduces throughput for small transfers, but yields better round-trip times. The socket buffer size was set to 64 Kbytes<sup>1</sup>, as this also improves TCP/IP round-trip latency. We tested TCP/IP and Fast Sockets for transfers up to 64 Kbytes; Active Messages only supports 4 Kbyte messages.

Figures 3 and 4 present the results of the round-trip microbenchmark. Fast Sockets achieves low latency round-trip times, especially for small packets. Round-trip time scales linearly with increasing transfer size, reflecting the time spent in moving the data to the network card. There is a "hiccup" at 4096 bytes on the Solaris platform, which is the maximum packet size for Active Messages. This occurs because Fast Sockets' fragmentation algorithm attempts to balance packet sizes for better round-trip and bandwidth characteristics. Fast Sockets' overhead is low, staying relatively constant at 25–30 microseconds (over that of Active Messages).

Table 1 shows the results of a least-squares linear regression analysis of the Solaris round-trip microbenchmark. We show  $t_0$ , the estimated cost for a

<sup>1</sup>TCP/IP is limited to a maximum buffer size of 56 Kbytes.

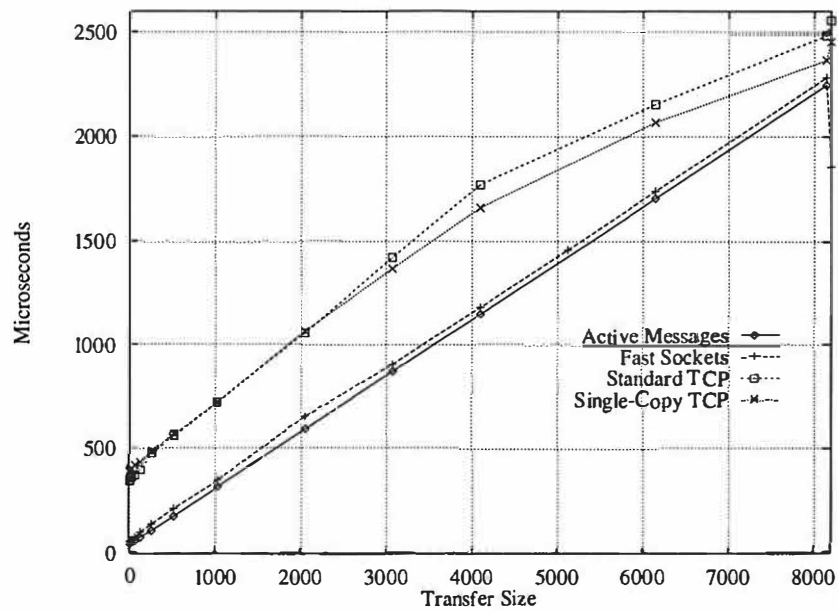


Figure 3: Round-trip performance for Fast Sockets, Active Messages, and TCP for the HP/UX/Medusa platform. Active Messages for HP/UX cannot transfer more than 8140 bytes at a time.

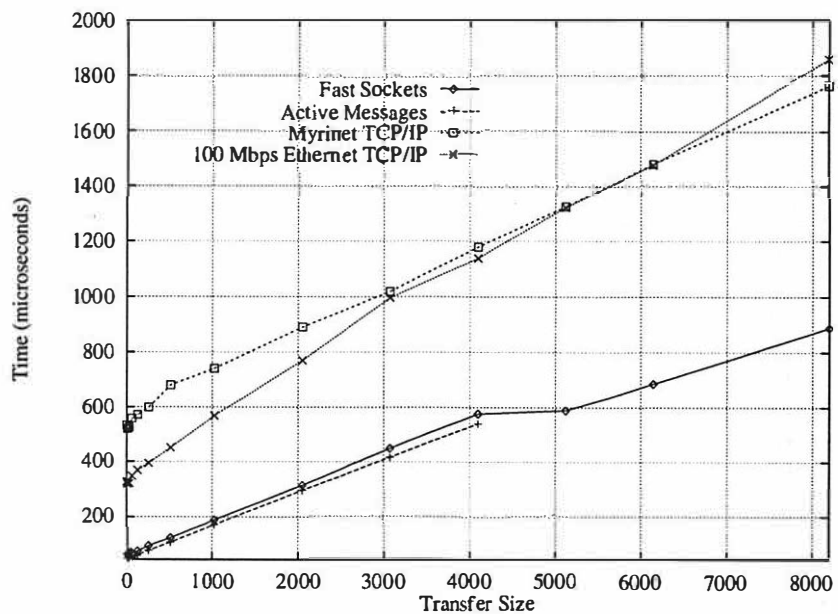


Figure 4: Round-trip performance for Fast Sockets, Active Messages, and TCP on the Solaris/Myrinet platform. Active Messages for Myrinet can only transfer up to 4096 bytes at a time.

0-byte packet, and the marginal cost for each additional data byte. Surprisingly, Fast Sockets' cost-per-byte appears to be lower than that of Active Messages. This is because the Fast Sockets per-byte cost is reported for a 64K range of transfers while Active Messages' per-byte cost is for a 4K range. The Active Messages test minimizes overhead by not implementing in-order delivery, which means only one packet can be outstanding at a time. Both Fast Sockets and TCP/IP provide in-order delivery, which enables data packets to be pipelined through the network and thus achieve a lower per-byte cost. The per-byte costs of Fast Sockets for small packets (less than 1 Kbyte) are slightly higher than Active Messages. While TCP's long-term per-byte cost is only about 15% higher than that of Fast Sockets, its performance for small packets is much worse, with per-byte costs twice that of Fast Sockets and startup costs 5–10 times higher.

Another surprising element of the analysis is that the overall  $t_0$  and  $t_1$  for small packets is very different, especially for Fast Sockets and Myrinet TCP/IP. Both protocols pipeline packets, which lowers round-trip latencies for multi-packet transfers. This causes a non-linear round-trip latency function, yielding different estimates of  $t_0$  for single-packet and multi-packet transfers.

#### 4.2.2 Bandwidth

Our bandwidth microbenchmark does 500 `send()` calls of a given size, and then waits for a response. This is repeated until a 95% confidence interval is obtained. As with the round-trip microbenchmark, the TCP/IP and Fast Sockets measurements were derived from the same program and Active Messages results were obtained using the same algorithm, but different code. Again, we used the TCP/IP `TCP_NODELAY` option to force immediate packet transmission, and a 64 Kbyte socket buffer.

Results for the bandwidth microbenchmark are shown in Figures 5 and 6. Fast Sockets is able to realize most of the available bandwidth of the network. On the UltraSPARC, the SBus is the limiting factor, rather than the network, with a maximum throughput of about 45 MB/s. Of this, Active Messages exposes 35 MB/s to user applications. Fast Sockets can realize about 90% of the Active Messages bandwidth, losing the rest to memory movement. Myrinet's TCP/IP only realizes 90% of the Fast Sockets bandwidth, limited by its high processing overhead.

Table 2 presents the results of a least-squares fit of the bandwidth curves to the equation

$$y = \frac{r_{\infty}x}{n_{\frac{1}{2}} + x}$$

| Layer                  | $r_{\infty}$<br>(MB/s) | $n_{\frac{1}{2}}$<br>(bytes) |
|------------------------|------------------------|------------------------------|
| Fast Sockets           | 32.9                   | 441                          |
| Active Messages        | 39.2                   | 372                          |
| TCP/IP (Myrinet)       | 29.6                   | 2098                         |
| TCP/IP (Fast Ethernet) | 11.1                   | 530                          |

Table 2: Least-squares regression analysis of the Solaris bandwidth microbenchmark.  $r_{\infty}$  is the maximum bandwidth of the network and is measured in megabytes per second. The half-power point ( $n_{\frac{1}{2}}$ ) is the packet size that delivers half of the maximum throughput, and is reported in bytes. TCP/IP was run with the `TCP_NODELAY` option, which attempts to transmit packets as soon as possible rather than coalescing data together.

which describes an idealized bandwidth curve.  $r_{\infty}$  is the theoretical maximum bandwidth realizable by the communications layer, and  $n_{\frac{1}{2}}$  is the *half-power point* of the curve. The half-power point is the transfer size at which the communications layer can realize half of the maximum bandwidth. A lower half-power point means a higher percentage of packets that can take advantage of the network's bandwidth. This is especially important given the frequency of small messages in network traffic. Fast Sockets' half-power point is only 18% larger than that of Active Messages, at 441 bytes. Myrinet TCP/IP realizes a maximum bandwidth 10% less than Fast Sockets but has a half-power point four times larger. Consequently, even though both protocols can realize much of the available network bandwidth, TCP/IP needs much larger packets to do so, reducing its usefulness for many applications.

#### 4.2.3 netperf and ttcp

Two commonly used microbenchmarks for evaluating network software are `netperf` and `ttcp`. Both of these benchmarks are primarily designed to test throughput, although `netperf` also includes a test of request-response throughput, measured in transactions/second.

We used a version 1.2 of `ttcp`, modified to work under Solaris, and `netperf` 2.11 for testing. The throughput results are shown in Figure 7, and our analysis is in Table 3.

A curious result is that the half-power points for `ttcp` and `netperf` are substantially lower for TCP/IP than on our bandwidth microbenchmark. One reason for this is that the maximum throughput for

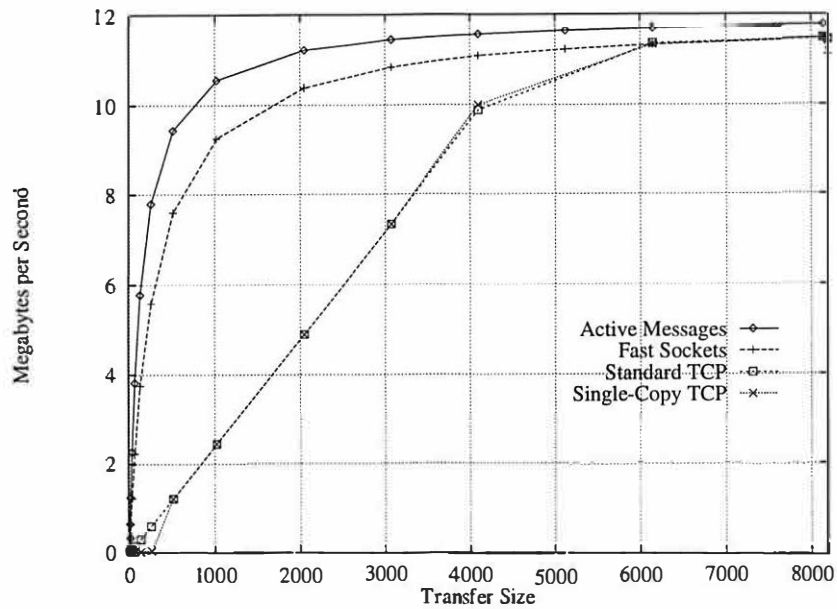


Figure 5: Observed bandwidth for Fast Sockets, TCP, and Active Messages on HP/UX with the Medusa FDDI network interface. The memory copy bandwidth of the HP735 is greater than the FDDI network bandwidth, so Active Messages and Fast Sockets can both realize close to the full bandwidth of the network.

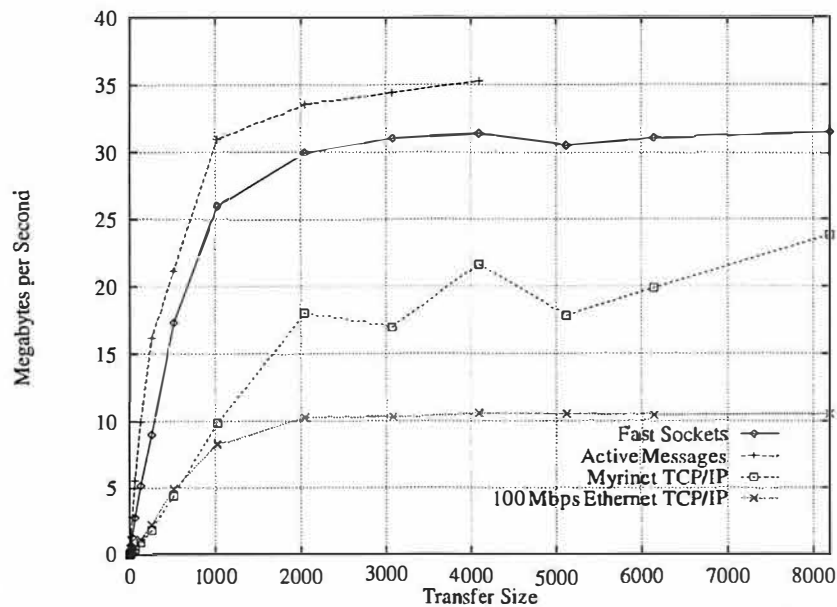


Figure 6: Observed bandwidth for Fast Sockets, TCP, and Active Messages on Solaris using the Myrinet local-area network. The bus limits the maximum throughput to 45MB/s. Fast Sockets is able to realize much of the available bandwidth of the network because of receive posting.

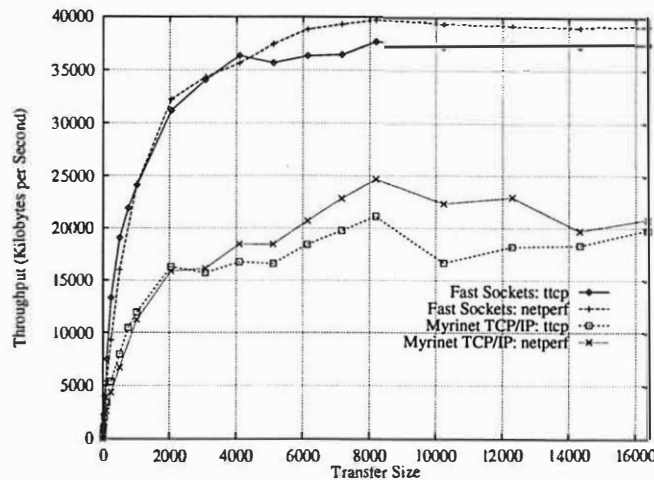


Figure 7: Ttcp and netperf bandwidth measurements on Solaris for Fast Sockets and Myrinet TCP/IP.

| Program/<br>Communication<br>Layer | $r_{\infty}$<br>(MB/s) | $n_{1/2}$<br>(bytes) |
|------------------------------------|------------------------|----------------------|
| ttcp                               |                        |                      |
| Fast Sockets                       | 38.57                  | 560.7                |
| Myrinet TCP/IP                     | 19.59                  | 687.5                |
| netperf                            |                        |                      |
| Fast Sockets                       | 41.57                  | 785.7                |
| Myrinet TCP/IP                     | 23.72                  | 1189                 |

Table 3: Least-squares regression analysis of the ttcp and netperf microbenchmarks. These tests were run on the Solaris 2.5.1/Myrinet platform. TCP/IP half-power point measures are lower than in Table 2 because both ttcp and netperf attempt to improve small-packet bandwidth at the price of small-packet latency.

TCP/IP is only about 50–60% that of Fast Sockets. Another reason is that TCP defaults to batching small data writes in order to maximize throughput (the *Nagle algorithm*) [Nagle 1984], and these tests do not disable the algorithm (unlike our microbenchmarks); however, this behavior trades off small-packet bandwidth for higher round-trip latencies, as data is held at the sender in an attempt to coalesce data.

The netperf microbenchmark also has a “request-response” test, which reports the number of transactions per second a communications stack is capable of for a given request and response size. There are two permutations of this test, one using an existing connection and one that establishes a connection every time; the latter closely mimics the

behavior of protocols such as HTTP. The results of these tests are reported in transactions per second and shown in Figure 8.

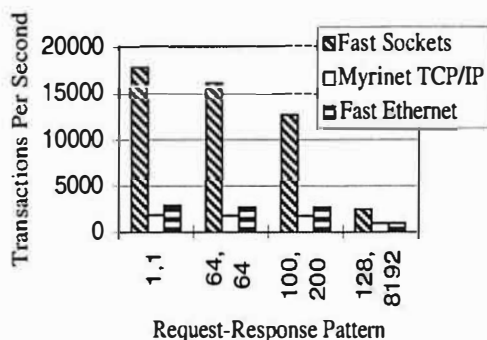
The request-response test shows Fast Sockets’ Achilles heel: its connection mechanism. While Fast Sockets does better than TCP/IP for request-response behavior with a constant connection (Figure 8(a)), introducing connection startup costs (Figure 8(b)) reduces or eliminates this advantage dramatically. This points out the need for an efficient, high-speed connection mechanism.

### 4.3 Applications

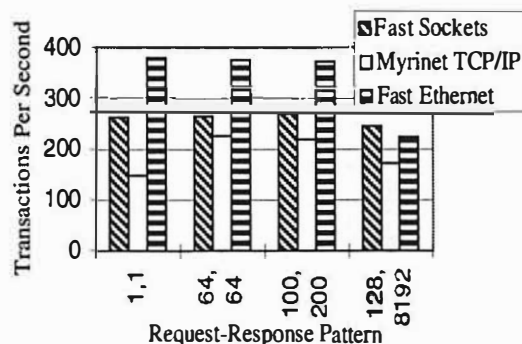
Microbenchmarks are useful for evaluating the raw performance characteristics of a communications implementation, but raw performance does not express the utility of a communications layer. Instead, it is important to characterize the difficulty of integrating the communications layer with existing applications, and the performance improvements realized by those applications. This section examines how well Fast Sockets supports the real-life demands of a network application.

#### 4.3.1 File Transfer

File transfer is traditionally considered a bandwidth-intensive application. However, the FTP protocol that is commonly used for file transfer still has a request-response nature. Further, we wanted to see what improvements in performance, if any, would be realized by using Fast Sockets for an application it was not intended for.



(a) Request-Response Performance



(b) Connection-Request-Response Performance

Figure 8: Netperf measurements of request-response transactions-per-second on the Solaris platform, for a variety of packet sizes. Connection costs significantly lower Fast Sockets' advantage relative to Myrinet TCP/IP, and render it slower than Fast Ethernet TCP/IP. This is because the current version of Fast Sockets uses the TCP/IP connection establishment mechanism.

We used the NcFTP ftp client (`ncftp`), version 2.3.0, and the Washington University ftp server (`wu-ftpd`), version 2.4.2. Because Fast Sockets currently does not support `fork()`, we modified `wu-ftpd` to wait for and accept incoming connections rather than be started from the `inetd` Internet server daemon.

Our FTP test involved the transfer of a number of ASCII files, of various sizes, and reporting the elapsed time and realized bandwidth as reported by the FTP client. On both machines, files were stored in an in-memory filesystem, to avoid the bandwidth limitations imposed by the disk.

The relative throughput for the Fast Sockets and TCP/IP versions of the FTP software is shown in Figure 9. Surprisingly, Fast Sockets and TCP/IP have roughly comparable performance for small files (1 byte to 4K bytes). This is due to the expense of connection setup — every FTP transfer involves the creation and destruction of a data connection. For mid-sized transfers, between 4 Kbytes and 2 Mbytes, Fast Sockets obtains considerably better bandwidth than normal TCP/IP. For extremely large transfers, both TCP/IP and Fast Sockets can realize a significant fraction of the network's bandwidth.

## 5 Related Work

Improving communications performance has long been a popular research topic. Previous work has focused on protocols, protocol and infrastructure implementations, and the underlying network device soft-

ware.

The VMTP protocol [Cheriton & Williamson 1989] attempted to provide a general-purpose protocol optimized for small packets and request-response traffic. It performed quite well for the hardware it was implemented on, but never became widely established; VMTP's design target was request-response and bulk transfer traffic, rather than the byte stream and datagram models provided by the TCP/IP protocol suite. In contrast, Fast Sockets provides the same models as TCP/IP and maintains application compatibility.

Other work [Clark et al. 1989, Watson & Mamrak 1987] argued that protocol implementations, rather than protocol designs, were to blame for poor performance, and that efficient implementations of general-purpose protocols could do as well as or better than special-purpose protocols for most applications. The measurements made in [Kay & Pasquale 1993] lend credence to these arguments; they found that memory operations and operating system overheads played a dominant role in the cost of large packets. For small packets, however, protocol costs were significant, amounting for up to 33% of processing time for single-byte messages.

The concept of reducing infrastructure costs was explored further in the *x-kernel* [Hutchinson & Peterson 1991, Peterson 1993], an operating system designed for high-performance communications. The original, stand-alone version of the *x-kernel* performed significantly better at communication tasks than did BSD Unix on the same hardware (Sun 3's),

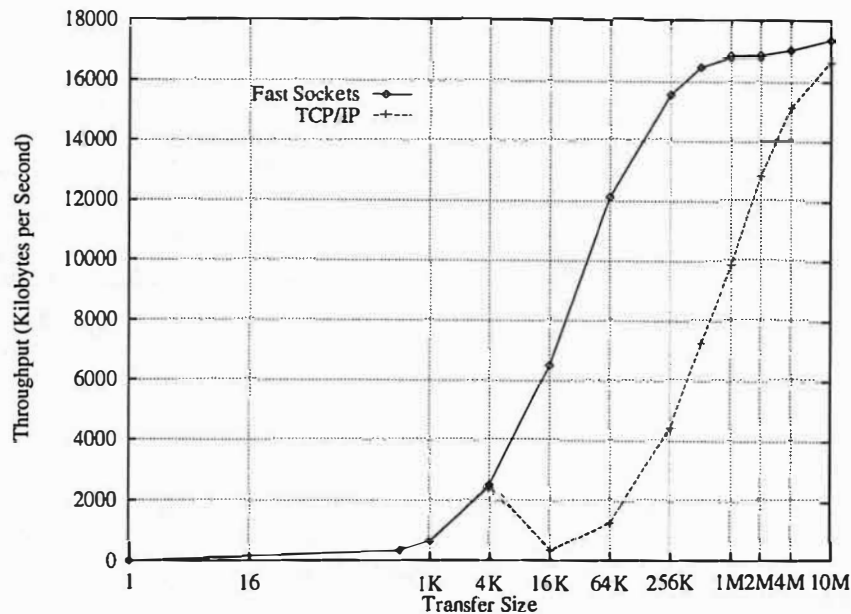


Figure 9: Relative throughput realized by Fast Sockets and TCP/IP versions of the FTP file transfer protocol. Connection setup costs dominate the transfer time for small files and the network transport serves as a limiting factor for large files. For mid-sized files, Fast Sockets is able to realize much higher bandwidth than TCP/IP.

using similar implementations of the communications protocols. Later work [Druschel et al. 1994, Pagels et al. 1994, Druschel et al. 1993, Druschel & Peterson 1993] focused on hardware design issues relating to network communication and the use of software techniques to exploit hardware features. Key contributions from this work were the concepts of *application device channels* (ADC), which provide protected user-level access to a network device, and *fbufs*, which provide a mechanism for rapid transfer of data from the network subsystem to the user application. While Active Messages provides the equivalent of an ADC for Fast Sockets, *fbufs* are not needed, as receive posting allows for data transfer directly into the user application.

Recently, the development of a zero-copy TCP stack in Solaris [Chu 1996] aggressively utilized hardware and operating system features such as direct memory access (DMA), page re-mapping, and copy-on-write pages to improve communications performance. To take full advantage of the zero-copy stack, user applications had to use page-aligned buffers and transfer sizes larger than a page. Because of these limitations, the designers focused on improving realized throughput, instead of small message latency, which Fast Sockets addresses. The resulting system achieved 32 MB/s throughput on a similar network but with a slower processor. This throughput was

achieved for large transfers (16K bytes), not the small packets that make up the majority of network traffic. This work required a thorough understanding of the Solaris virtual memory subsystem, and changes to the operating system kernel; Fast Sockets is an entirely user-level solution.

An alternative to reducing internal operating system costs is to bypass the operating system altogether, and use either in a user-level library (like Fast Sockets) or a separate user-level server. Mach 3.0 used the latter approach [Forin et al. 1991], which yielded poor networking performance [Maeda & Bershad 1992]. Both [Maeda & Bershad 1993a] and [Thekkath et al. 1993] explored building TCP into a user-level library linked with existing applications. Both systems, however, attempted only to match in-kernel performance, rather than better it. Further, both systems utilized in-kernel facilities for message transmission, limiting the possible performance improvement. Edwards and Muir [Edwards & Muir 1995] attempted to build an entirely user-level solution, but utilized a TCP stack that had been built for the HP/UX kernel. Their solution replicated the organization of the kernel at user-level with worse performance than the in-kernel TCP stack.

[Kay & Pasquale 1993] showed that interfacing to the network card itself was a major cost for small packets. Recent work has focused on reduc-

ing this portion of the protocol cost, and on utilizing the message coprocessors that are appearing on high-performance network controllers such as Myrinet [Seitz 1994]. Active Messages [von Eicken et al. 1992] is the base upon which Fast Sockets is built and is discussed above. Illinois Fast Messages [Pakin et al. 1995] provided an interface similar to that of previous versions of Active Messages, but did not allow processes to share the network. Remote Queues [Brewer et al. 1995] provided low-overhead communications similar to that of Active Messages, but separated the arrival of messages from the invocation of handlers.

The SHRIMP project implemented a stream sockets layer that uses many of the same techniques as Fast Sockets [Damianakis et al. 1996]. SHRIMP supports communication via shared memory and the execution of handler functions on data arrival. The SHRIMP network had hardware latencies (4–9  $\mu$ s one-way) much lower than the Fast Sockets Myrinet, but its maximum bandwidth (22 MB/s) was also lower than that of the Myrinet [Felten et al. 1996]. It used a custom-designed network interface for its memory-mapped communication model. The interface provided in-order, reliable delivery, which allowed for extremely low overheads (7  $\mu$ s over the hardware latency); Fast Sockets incurs substantial overhead to ensure in-order delivery. Realized bandwidth of SHRIMP sockets was about half the raw capacity of the network because the SHRIMP communication model used sender-based memory management, forcing data transfers to indirect through the socket buffer. The SHRIMP communication model also deals poorly with non-word-aligned data, which required programming complexity to work around; Fast Sockets transparently handles this un-aligned data without extra data structures or other difficulties in the data path.

U-Net [von Eicken et al. 1995] is a user-level network interface developed at Cornell. It virtualized the network interface, allowing multiple processes to share the interface. U-Net emphasized improving the implementation of existing communications protocols whereas Fast Sockets uses a new protocol just for local-area use. A version of TCP (U-Net TCP) was implemented for U-Net; this protocol stack provided the full functionality of the standard TCP stack. U-Net TCP was modified for better performance; it succeeded in delivering the full bandwidth of the underlying network but still imposed more than 100 microseconds of packet processing overhead relative to the raw U-Net interface.

## 6 Conclusions

In this paper we have presented Fast Sockets, a communications interface which provides low-overhead, low-latency and high-bandwidth communication on local-area networks using the familiar Berkeley Sockets interface. We discussed how current implementations of the TCP/IP suite have a number of problems that contribute to poor latencies and mediocre bandwidth on modern high-speed networks, and how Fast Sockets was designed to directly address these shortcomings of TCP/IP implementations. We showed that this design delivers performance that is significantly better than TCP/IP for small transfers and at least equivalent to TCP/IP for large transfers, and that these benefits can carry over to real-life programs in everyday usage.

An important contributor to Fast Socket's performance is receive posting, which utilizes socket-layer information to influence the delivery actions of layers farther down the protocol stack. By moving destination information down into lower layers of the protocol stack, Fast Sockets bypasses copies that were previously unavoidable.

Receive posting is an effective and useful tool for avoiding copies, but its benefits vary greatly depending on the data transfer mechanism of the underlying transport layer. Sender-based memory management schemes impose high synchronization costs on messaging layers such as Sockets, which can affect realized throughput. A receiver-based system reduces the synchronization costs of receive posting and enables high throughput communication without significantly affecting round-trip latency.

In addition to receive posting, Fast Sockets also collapses multiple protocol layers together and reduces the complexity of network buffer management. The end result of combining these techniques is a system which provides high-performance, low-latency communication for existing applications.

## Availability

Implementations of Fast Sockets are currently available for Generic Active Messages 1.0 and 1.5, and for Active Messages 2.0. The software and current information about the status of Fast Sockets can be found on the Web at <http://now.cs.berkeley.edu/Fast-comm/fastcomm.html>.

## Acknowledgements

We would like to thank John Schimmel, our shepherd, Douglas Ghormley, Neal Cardwell, Kevin Skadron, Drew Roselli, and Eric Anderson for their advice and comments in improving the presentation of this paper. We would also like to thank Rich Martin, Remzi Arpacı, Amin Vahdat, Brent Chun, Alan Mainwaring, and the other members of the Berkeley NOW group for their suggestions, ideas, and help in putting this work together.

## References

- [Abbott & Peterson 1993] M. B. Abbott and L. L. Peterson. "Increasing network throughput by integrating protocol layers." *IEEE/ACM Transactions on Networking*, 1(5):600-610, October 1993.
- [Amer et al. 1987] P. D. Amer, R. N. Kumar, R. bin Kao, J. T. Phillips, and L. N. Cassel. "Local area broadcast network measurement: Traffic characterization." In *Spring COMPCON*, pp. 64-70, San Francisco, California, February 1987.
- [Banks & Prudence 1993] D. Banks and M. Prudence. "A high-performance network architecture for a PA-RISC workstation." *IEEE Journal on Selected Areas in Communications*, 11(2):191-202, February 1993.
- [Birrell & Nelson 1984] A. D. Birrell and B. J. Nelson. "Implementing remote procedure calls." *ACM Transactions on Computer Systems*, 2(1):39-59, February 1984.
- [Blumrich et al. 1994] M. A. Blumrich, K. Li, R. D. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg. "Virtual memory mapped network interface for the SHRIMP multicomputer." In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pp. 142-153, Chicago, IL, April 1994.
- [Brewer et al. 1995] E. A. Brewer, F. T. Chong, L. T. Liu, S. D. Sharma, and J. D. Kubiatowicz. "Remote Queues: Exposing message queues for optimization and atomicity." In *Proceedings of SPAA '95*, Santa Barbara, CA, June 1995.
- [Buzzard et al. 1996] G. Buzzard, D. Jacobson, M. Mackey, S. Marovich, and J. Wilkes. "An implementation of the Hamlyn sender-managed interface architecture." In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pp. 245-259, Seattle, WA, October 1996.
- [Caceres et al. 1991] R. Caceres, P. B. Danzig, S. Jamin, and D. J. Mitzel. "Characteristics of wide-area TCP/IP conversations." In *Proceedings of ACM SIGCOMM '91*, September 1991.
- [Cheriton & Williamson 1987] D. R. Cheriton and C. L. Williamson. "Network measurement of the VMTP request-response protocol in the V distributed system." In *Proceedings of SIGMETRICS '87*, pp. 216-225, Banff, Alberta, Canada, May 1987.
- [Cheriton & Williamson 1989] D. Cheriton and C. Williamson. "VMTP: A transport layer for high-performance distributed computing." *IEEE Communications*, pp. 37-44, June 1989.
- [Chu 1996] H.-K. J. Chu. "Zero-copy TCP in Solaris." In *Proceedings of the 1996 USENIX Annual Technical Conference*, San Diego, CA, January 1996.
- [Claffy et al. 1992] K. C. Claffy, G. C. Polyzos, and H.-W. Braun. "Traffic characteristics of the T1 NSFNET backbone." Technical Report CS92-270, University of California, San Diego, July 1992.
- [Clark & Tennenhouse 1990] D. D. Clark and D. L. Tennenhouse. "Architectural considerations for a new generation of protocols." In *Proceedings of SIGCOMM '90*, pp. 200-208, Philadelphia, Pennsylvania, September 1990.
- [Clark 1982] D. D. Clark. "Modularity and efficiency in protocol implementation." Request For Comments 817, IETF, July 1982.
- [Clark et al. 1989] D. D. Clark, V. Jacobson, J. Romkey, and H. Salwen. "An analysis of TCP processing overhead." *IEEE Communications*, pp. 23-29, June 1989.
- [Crowcroft et al. 1992] J. Crowcroft, I. Wakeman, Z. Wang, and D. Sirovica. "Is layering harmful?" *IEEE Network*, 6(1):20-24, January 1992.
- [Culler et al. 1994] D. E. Culler, K. Keeton, L. T. Liu, A. Mainwaring, R. P. Martin, S. H. Rodrigues, and K. Wright. *The Generic Active Message Interface Specification*, February 1994.
- [Dahlin et al. 1994] M. D. Dahlin, R. Y. Wang, D. A. Patterson, and T. E. Anderson. "Cooperative caching: Using remote client memory to improve file system performance." In *Proceedings of the First Conference on Operating Systems Design and Implementation (OSDI)*, pp. 267-280, October 1994.
- [Damianakis et al. 1996] S. N. Damianakis, C. Dubnicki, and E. W. Felten. "Stream sockets on SHRIMP." Technical Report TR-513-96, Princeton University, Princeton, NJ, October 1996.
- [de Prycker 1993] M. de Prycker. *Asynchronous Transfer Mode: Solution for Broadband ISDN*. Ellis Horwood Publishers, second edition, 1993.

- [Druschel & Peterson 1993] P. Druschel and L. L. Peterson. "Fbufs: A high-bandwidth cross-domain data transfer facility." In *Proceedings of the Fourteenth Annual Symposium on Operating Systems Principles*, pp. 189–202, Asheville, NC, December 1993.
- [Druschel et al. 1993] P. Druschel, M. B. Abbott, M. A. Pagels, and L. L. Peterson. "Network subsystem design: A case for an integrated data path." *IEEE Network (Special Issue on End-System Support for High Speed Networks)*, 7(4):8–17, July 1993.
- [Druschel et al. 1994] P. Druschel, L. L. Peterson, and B. S. Davie. "Experiences with a high-speed network adapter: A software perspective." In *Proceedings of ACM SIGCOMM '94*, August 1994.
- [Edwards & Muir 1995] A. Edwards and S. Muir. "Experiences in implementing a high performance TCP in user-space." In *ACM SIGCOMM '95*, Cambridge, MA, August 1995.
- [Feldmeier 1986] D. C. Feldmeier. "Traffic measurement on a token-ring network." In *Proceedings of the 1986 Computer Networking Conference*, pp. 236–243, November 1986.
- [Felten et al. 1996] E. W. Felten, R. D. Alpert, A. Bilas, M. A. Blumrich, D. W. Clark, S. N. Damiakis, C. Dubnicki, L. Iftode, and K. Li. "Early experience with message-passing on the SHRIMP multicomputer." In *Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA '96)*, pp. 296–307, Philadelphia, PA, May 1996.
- [Forin et al. 1991] A. Forin, D. Golub, and B. N. Bershad. "An I/O system for Mach 3.0." In *Proceedings of the USENIX Mach Symposium*, pp. 163–176, Monterey, CA, November 1991.
- [Gusella 1990] R. Gusella. "A measurement study of diskless workstation traffic on an Ethernet." *IEEE Transactions on Communications*, 38(9):1557–1568, September 1990.
- [Hutchinson & Peterson 1991] N. Hutchinson and L. L. Peterson. "The x-kernel: An architecture for implementing network protocols." *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.
- [Kay & Pasquale 1993] J. Kay and J. Pasquale. "The importance of non-data-touching overheads in TCP/IP." In *Proceedings of the 1993 SIGCOMM*, pp. 259–268, San Francisco, CA, September 1993.
- [Keeton et al. 1995] K. Keeton, D. A. Patterson, and T. E. Anderson. "LogP quantified: The case for low-overhead local area networks." In *Hot Interconnects III*, Stanford University, Stanford, CA, August 1995.
- [Kleinrock & Naylor 1974] L. Kleinrock and W. E. Naylor. "On measured behavior of the ARPA network." In *AFIPS Proceedings*, volume 43, pp. 767–780, 1974.
- [Liu & Culler 1995] L. T. Liu and D. E. Culler. "Evaluation of the Intel Paragon on Active Message communication." In *Proceedings of the 1995 Intel Supercomputer Users Group Conference*, June 1995.
- [Maeda & Bershad 1992] C. Maeda and B. N. Bershad. "Networking performance for microkernels." In *Proceedings of the Third Workshop on Workstation Operating Systems*, pp. 154–159, 1992.
- [Maeda & Bershad 1993a] C. Maeda and B. Bershad. "Protocol service decomposition for high-performance networking." In *Proceedings of the Fourteenth Symposium on Operating Systems Principles*, pp. 244–255, Asheville, NC, December 1993.
- [Maeda & Bershad 1993b] C. Maeda and B. Bershad. "Service without servers." In *Workshop on Workstation Operating Systems IV*, October 1993.
- [Mainwaring & Culler 1995] A. Mainwaring and D. Culler. *Active Messages: Organization and Applications Programming Interface*, September 1995.
- [Martin 1994] R. P. Martin. "HPAM: An Active Message layer for a network of HP workstations." In *Hot Interconnects II*, pp. 40–58, Stanford University, Stanford, CA, August 1994.
- [Nagle 1984] J. Nagle. "Congestion control in IP/TCP internetworks." Request For Comments 896, Network Working Group, January 1984.
- [Pagels et al. 1994] M. A. Pagels, P. Druschel, and L. L. Peterson. "Cache and TLB effectiveness in processing network I/O." Technical Report 94-08, University of Arizona, March 1994.
- [Pakin et al. 1995] S. Pakin, M. Lauria, and A. Chien. "High-performance messaging on workstations: Illinois Fast Messages(FM) for Myrinet." In *Supercomputing '95*, San Diego, CA, 1995.
- [Peterson 1993] L. L. Peterson. "Life on the OS/network boundary." *Operating Systems Review*, 27(2):94–98, April 1993.
- [Postel 1981a] J. Postel. "Internet protocol." Request For Comments 791, IETF Network Working Group, September 1981.
- [Postel 1981b] J. Postel. "Transmission control protocol." Request For Comments 793, IETF Network Working Group, September 1981.
- [Postel 1981c] J. Postel. "User datagram protocol." Request For Comments 768, IETF Network Working Group, August 1981.

- [Schoch & Hupp 1980] J. F. Schoch and J. A. Hupp. "Performance of an Ethernet local network." *Communications of the ACM*, 23(12):711–720, December 1980.
- [Seitz 1994] C. Seitz. "Myrinet: A gigabit-per-second local area network." In *Hot Interconnects II*, Stanford University, Stanford, CA, August 1994.
- [Stevens 1990] W. R. Stevens. *UNIX Network Programming*. Prentice Hall, Englewood Cliffs, NJ, 1990.
- [Thekkath et al. 1993] C. A. Thekkath, T. Nguyen, E. Moy, and E. D. Lazowska. "Implementing network protocols at user-level." *IEEE/ACM Transactions on Networking*, pp. 554–565, October 1993.
- [von Eicken et al. 1992] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauer. "Active Messages: A mechanism for integrated communication and computation." In *Proceedings of the Nineteenth ISCA*, Gold Coast, Australia, May 1992.
- [von Eicken et al. 1995] T. von Eicken, A. Basu, V. Buch, and W. Vogels. "U-Net: A user-level network interface for parallel and distributed computing." In *Proceedings of the Fifteenth SOSP*, pp. 40–53, Copper Mountain, CO, December 1995.
- [Watson & Mamrak 1987] R. M. Watson and S. A. Mamrak. "Gaining efficiency in transport services by appropriate design and implementation choices." *ACM Transactions on Computer Systems*, 5(2):97–120, May 1987.

# An Analytical Approach to File Prefetching

Hui Lei and Dan Duchamp

*Computer Science Department  
Columbia University  
New York, NY 10027  
{lei, djd}@cs.columbia.edu*

## Abstract

File prefetching is an effective technique for improving file access performance. In this paper, we present a file prefetching mechanism that is based on on-line analytic modeling of interesting system events and is transparent to higher levels. The mechanism, incorporated into a client's file cache manager, seeks to build semantic structures that capture the intrinsic correlations between file accesses. It then heuristically uses these structures to represent distinct file usage patterns and exploits them to prefetch files from a file server. We show results of a simulation study and of a working implementation. Measurements suggest that our method can predict future file accesses with an accuracy around 90%, that it can reduce cache miss rate by up to 47% and application latency by up to 40%. Our method imposes little overhead, even under antagonistic circumstances.

## 1 Introduction

This paper reports the effectiveness of a predictive file prefetching technique that operates automatically and without any sort of information supplied by applications or users. The technique, which is incorporated into the client's cache manager, makes extra requests to the server, hopefully in advance of the actual need for the data. Prefetched data is then placed in the client's cache.

We hypothesize that there is pronounced regularity in file access patterns and relatively simple algorithms can identify an access pattern and quickly spot it when it re-emerges later during another run of the application. In particular, we build semantic data structures, called *access trees*, that capture potentially useful information concerning the interrelationships and dependencies between files. An access tree for a program records all the files referenced dur-

ing one execution of the program. For each program, we maintain a number of access trees in virtual memory that represent distinct file usage patterns. When a program is re-executed, we compare the access tree being formed by current activity against saved access trees to determine which usage pattern, if any, is recurring; we then prefetch the files remaining in the saved access tree.

File prefetching brings two major advantages. First, applications run faster because they hit more in the file cache. Second, there is less "burst" load placed on the network because prefetching is done only when there is network bandwidth available rather than on demand. On the other hand, there are two main costs of prefetching. The first is the CPU cycles expended by the client in determining when and what to prefetch. Cycles are spent both on overhead in gathering the information necessary to make prefetch decisions, and on actually carrying out the prefetch. The second cost is the network bandwidth and server capacity wasted when prefetch decisions inevitably prove less than perfect.

We have conducted our study in the UNIX environment, which is ubiquitous in academia and the research community. It has been well recognized that on UNIX most files are accessed in their entirety and sequentially [19, 2]. We take advantage of this phenomenon in two ways. First, we effectively model file system events at the level of whole files, and look for access patterns across files. Second, if a file is predicted, we prefetch the initial portion of the file only, letting the standard sequential read-ahead mechanism bring in the rest when the file is demanded.

Section 2 details the prefetching mechanism. Section 3 reports results from a trace-driven simulation. Section 4 describes the implementation and presents some initial performance data. Section 5 discusses related work.

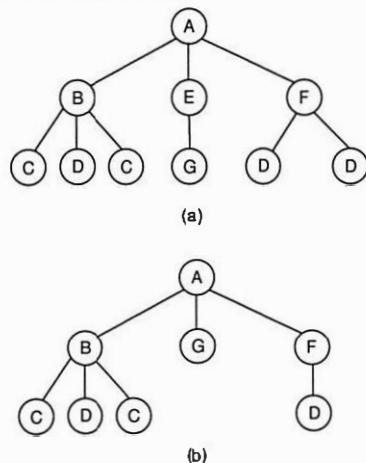


Figure 1: **Example Access Trees.** These graphs show the access trees generated from the events described in Section 1.2. Graph (a) shows the version before compression. Graph (b) that after both vertical and horizontal compression, assuming E is a shell program.

## 2 Mechanism

### 2.1 Data Abstraction: Access Tree

Every program can invoke other programs. In UNIX-style operating systems, this is usually realized by the executing program forking child processes, which in turn execute other programs. The programs may also open some data files. All the file references in an application can be formulated into a tree data structure, dubbed an access tree. Files (program files and data files) are the nodes, and an edge is drawn from parent A to child B if either (1) program A invokes program B or (2) program A opens data file B. The order of siblings reflects the chronology of file accesses.

Figure 1(a) depicts the access tree for an application A that includes the following activities:

1. Program A invokes program B
2. B opens data files C and D, in that order
3. B opens C again
4. A invokes programs E and F
5. F opens D
6. E invokes program G
7. F opens D again

An access tree is subject to two kinds of compression. Vertical compression draws edges through UNIX shells, effectively cutting them out of the access tree. This is necessary because of the role shells play as command interpreters. Shells are invoked in a variety of circumstances and can generate a large number of file usage patterns. Since we perform file prefetching for each program file in the access tree and it is infeasible to prefetch accurately for shells, we choose to ignore them. Horizontal compression removes consecutive accesses of the same data file. The detail of consecutive accesses offers no help in prefetching and can be safely omitted. However, non-consecutive accesses of the same data file are preserved for use in a later phase. Assuming program E is a shell in the previous example, Figure 1(b) shows the access tree after compressions.

An access tree describes the context in which file references are incurred in an application. We feel that this context information, if made available to the prefetching mechanism, would decrease the apparent randomness of file system events and permit more accurate predictions of future file references.

As an application proceeds, we construct an access tree for it by intercepting `fork`, `execve`, `open`, `chdir` and `exit` system calls. `Execve` and `open` calls deal with references of program files and data files respectively. `Fork` provides information on how processes, or program executions, are associated with each other. Information on `chdir` calls is used to resolve relative pathnames of files. The access tree is completed upon `exit` of the program execution. An access tree that is being constructed by current activity is called a *working tree*; a finished access tree that is saved to exemplify a file usage pattern is called a *pattern tree*.

### 2.2 The Prefetch Algorithm

We use a heuristic function `compatibility(w_tree, p_tree)` which returns an indication between 0 and 1 of the degree to which a working tree `w_tree` and a pattern tree `p_tree` resemble each other. When the function's value is above constant `MATCH_THRESHOLD`, we say that the two trees *match*, meaning that they are similar enough that they are considered to belong to the same access pattern. We shall explain the definition of `compatibility` function in more detail after we discuss the basic operation of the mechanism.

As mentioned earlier, a number of pattern trees are saved in virtual memory for each program; a working tree is constructed in the course of every program execution. Whenever a program references a file, a new child node is added in the working tree

for the program, and some analysis is performed to find out whether any saved pattern trees can be prefetched. Our analysis follows a simple guideline: if there is no previously prefetched pattern tree or the current working tree no longer matches the prefetched pattern tree, compute the compatibility of the working tree and each of the pattern trees for the program. If any match is found, then prefetch the pattern tree with the highest compatibility. If more than one pattern tree bears the highest compatibility, then prefetch the one most recently saved.

The above analysis is carried out for each executable file inside the working tree whenever the executable initiates a file reference. At this point, a pattern tree may have already been prefetched for the executable, either as the result of prefetch analysis incurred by earlier file accesses or in the form of a subtree of the larger tree we prefetched at a higher level. The executable can always prefetch another pattern tree, based on the analysis result. This effectively allows minor prefetch corrections to be made, reducing the cost of a bad guess.

Two complications may arise when the pattern tree selected for prefetching is large: prefetched files may be evicted from the cache before they are actually referenced, and the cache is considerably destroyed when the prefetching guess is bad. To diminish the extent of these problems, we place an upper limit (`PREFETCH_CAPACITY`) on the number of files from a pattern tree we shall prefetch at one time. When the pattern tree selected is too large, we prefetch only those files in the initial portion of the tree so that the prefetch limit is not exceeded. We also record the immediate child node of the pattern tree we prefetch last. When later the working tree extends to this child, we will prefetch the remaining portion of the pattern tree, if the latest working tree still matches it. We measure `PREFETCH_CAPACITY` in number of files, rather than in number of bytes, because such a measure goes well with the access tree structure, where each tree node represents a file. Further, as illustrated later, only the initial portion of a predicted file will be prefetched, hence the cost of an incorrect prediction is proportional to the number of prefetched files.

On program exit, we compare the newly completed working tree with the saved pattern trees. If it doesn't match any of the pattern trees, the working tree is saved as new information. Otherwise, it is substituted for the pattern tree that it matches best.

We set the two algorithmic parameters to the following values:

- `MATCH_THRESHOLD`: 0.4

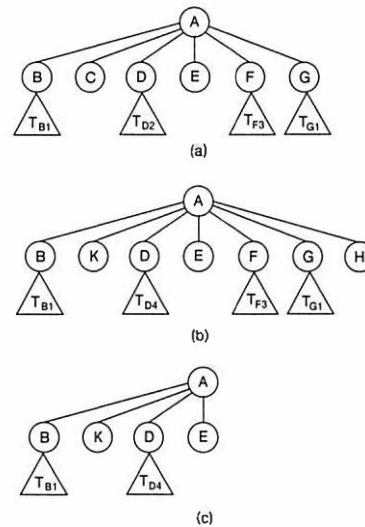


Figure 2: **Computation of Compatibility.** Graph (a) shows a pattern tree. Graph (b) shows a finished working tree, bearing a 0.575 compatibility with the given pattern tree. Graph (c) shows a unfinished working tree, so far bearing a 0.5 compatibility with the pattern tree; E is the pivot in the pattern tree, since it corresponds to the latest child node in the working tree being formed.

- `PREFETCH_CAPACITY`: 15

We have found that the behavior of our mechanism is insensitive to the value of these parameters. We shall illustrate this later.

### 2.3 Compatibility Computation

We now take a closer look at the `compatibility` function. This function, essentially a similarity metric between access trees, abstracts out the complexities of prefetch analysis and pattern tree maintenance. We illustrate the definition with the example in Figure 2. A pattern tree is shown in 2(a).

Let us first consider the case that the working tree is a finished one. We try to pair up the immediate child nodes in the working tree with identical child nodes in the pattern tree, preserving the order of the nodes. Recall that a child node can represent either an executable file, which may root another access tree, or a data file. For the two trees to be considered to describe the same file access pattern, we require that there be a one-to-one correspondence between all the executable files, but not all the data files. However, the same executable file may root a different access subtree in the working tree than in

the pattern tree. We define  $C_d$  as the percentage of data files that can be paired up,  $C_e$  as the percentage of pairs of executables that root the same access subtree. Intuitively,  $C_d$  suggests how “compatible” the data files are,  $C_e$  suggests how “compatible” the executable files are. We choose to use the average of the two values as the compatibility of the two trees in question.

Let us consider the finished working tree in Figure 2(b). There are three data files in the working tree and two in the pattern tree. Out of these five files, only the two appearances of E can be paired up, making  $C_d$  0.4. Three out of four executable pairs (B, F and G) root identical access subtrees, so  $C_e$  is 0.75. The compatibility is thus 0.575.

The case of an unfinished working tree is similar except that one child in the pattern tree is first determined to be the *pivot* node. The pivot corresponds to the most recently added child node in the working tree. Only the child nodes in the pattern tree that appear before the pivot are involved in compatibility computation. If the pattern tree is selected, we prefetch those files that follow the pivot in the sequence of pattern tree preordering, since those before the pivot probably have already been accessed. Given the unfinished working tree in Figure 2(c), node E is the pivot in the pattern tree.  $C_d$  and  $C_e$  are both 0.5, giving rise to a 0.5 compatibility. If we decide to prefetch this pattern tree, only the files in subtrees  $T_{F3}$  and  $T_{G1}$  will be prefetched.

Since the compatibility function is invoked often, it is important that it not be expensive. That is why we examine only immediate child nodes. The time complexity of the computation is proportional to the number of child nodes.

### 3 Simulation

Our initial assessments of the mechanism were obtained by trace-driven simulation.

#### 3.1 Method

We gathered several file traces [24] on SunOS 4.0.3c. This version of SunOS offers a “C2 secure computing facility” that includes the ability to produce a system call audit trail. Using this feature, we gathered three traces of a volunteer user performing his normal work activity over a period of two weeks. The first trace contains 25,441 invocations of previously enumerated system calls captured over 72 hours. The second trace contains 23,858 invocations captured over 52 hours, while the numbers for the third trace are 85,962 and 86, respectively. During these hours activity varied widely and included compilations, document production, data analysis and

display, large file searches, news reading, printing, and other operations.

Our simulated cache manager used an LRU replacement policy. It stepped through the traces, maintaining the cache in accordance with the user file accesses. Since our trace data lacks file sizes, we defined cache size by number of files. We varied the cache size and for each cache size, we compared the results of prefetching against those without prefetching.

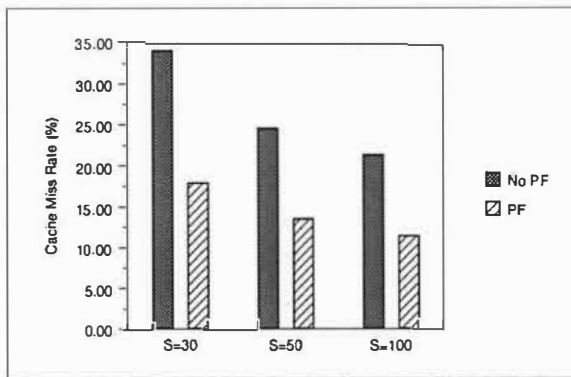
#### 3.2 Results

Our first metric was the cache miss rate. The results for the three traces appears in Figure 3. Prefetching delivers a substantially better miss rate with all cache sizes. The results are worse for the second trace because it includes a recursive directory traversal (i.e., the UNIX `find` program) over a hierarchy that includes thousands of files that are never accessed again.

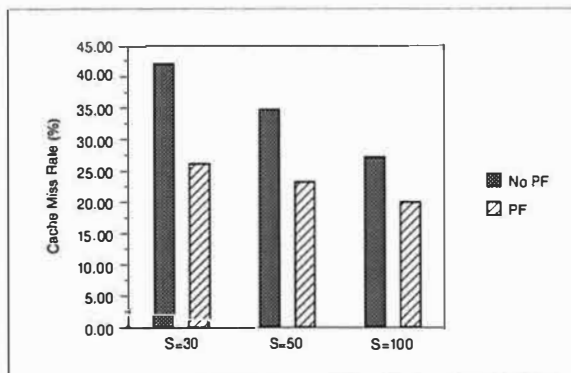
In addition, we monitored the cache behavior at a finer grain: how well our mechanism worked over each run of 500 accesses. At the end of each run, we checked to see whether prefetching had beaten no-prefetching by measuring the number of misses during that run. As shown in Table 1, prefetching won this comparison easily and consistently. This shows that our mechanism’s superiority is steady and stable, and not simply a result of a few exceptionally fruitful prefetch sequences. The few losses due to bad prefetching guesses are more than offset by the many wins.

Both Figure 3 and Table 1 indicate that the increased intelligence of our mechanism is more effective in smaller caches. This was expected. Consider that, in the extreme case of an infinitely large cache, any file appearing in any access tree is still in the cache. There is no room for improvement from any prefetching algorithm that is based solely on information about the past.

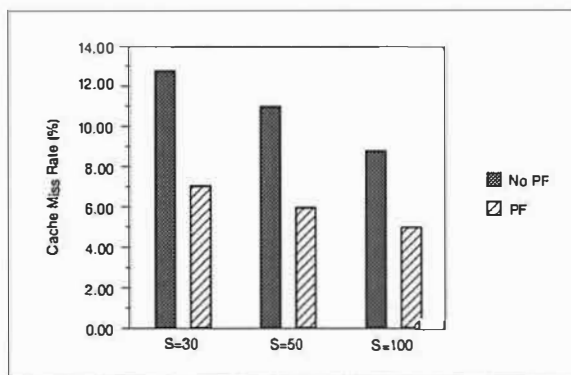
We measured the accuracy of our prefetch decisions, defined as the percentage of file access predictions that were actually used. We also calculated one overhead of the mechanism, i.e., the percentage of the file fetches that were initiated due to bad prefetch decisions. This reflects the network bandwidth and server capacity that were wasted. Table 2 shows these results. (We shall address another overhead, the CPU cycles expended by the prefetcher, in Section 4.) Because of the LRU policy, bigger caches give better accuracy results because prefetched entries have a better chance to survive cache entry replacements. Similarly, bigger caches also give better overhead results because bad predictions have a bet-



(a) Trace 1



(b) Trace 2



(c) Trace 3

**Figure 3: Cache Miss Rate.** These charts compare the accumulative cache miss rate with prefetching (PF) against that without prefetching (No PF). The comparison is performed under three different cache sizes (S), measured by number of files.

ter chance to have already existed in the cache; in such a case, no fetch is performed.

The *compatibility* function plays a critical role in our mechanism. The assumption underlying this

| Trace | Cache Size | Wins | Ties | Losses |
|-------|------------|------|------|--------|
| 1     | 30         | 31   | 6    | 0      |
|       | 50         | 29   | 6    | 2      |
|       | 100        | 23   | 12   | 2      |
| 2     | 30         | 31   | 4    | 0      |
|       | 50         | 30   | 4    | 1      |
|       | 100        | 23   | 9    | 3      |
| 3     | 30         | 88   | 45   | 8      |
|       | 50         | 82   | 54   | 5      |
|       | 100        | 69   | 65   | 7      |

**Table 1: A Finer-grained Comparison.** At the end of each run of 500 file accesses, we determined whether prefetching had beaten no-prefetching by measuring the number of cache misses during that run.

| Trace | Cache Size | Accuracy (%) | Overhead (%) |
|-------|------------|--------------|--------------|
| 1     | 30         | 94.15        | 5.13         |
|       | 50         | 95.17        | 4.61         |
|       | 100        | 96.04        | 3.83         |
| 2     | 30         | 88.09        | 8.60         |
|       | 50         | 90.92        | 6.93         |
|       | 100        | 93.07        | 5.34         |
| 3     | 30         | 90.68        | 12.51        |
|       | 50         | 93.40        | 9.63         |
|       | 100        | 95.36        | 7.25         |

**Table 2: Prefetching Accuracy and Overhead.** The *accuracy* is the percentage of the predictions that were actually used. The *overhead* is the percentage of the file fetches due to bad guesses.

function is that if the initial portions of two access trees bear a high compatibility, so will the two trees in their entirety. In order to test this assumption, we examined the *number of loads*, i.e., the total number of times we prefetched pattern trees. We further classified these loads as either *proper* or *improper*. A load became improper when a different pattern tree was selected by the prefetcher to take the place of the current one, or when the finished working tree did not match the pattern tree loaded. Otherwise, the load was proper. Our results in Figure 4 suggest that the compatibility function is effective.

Finally, we note that the algorithmic parameters should be stable: small variations in the settings must not produce large performance degradations.

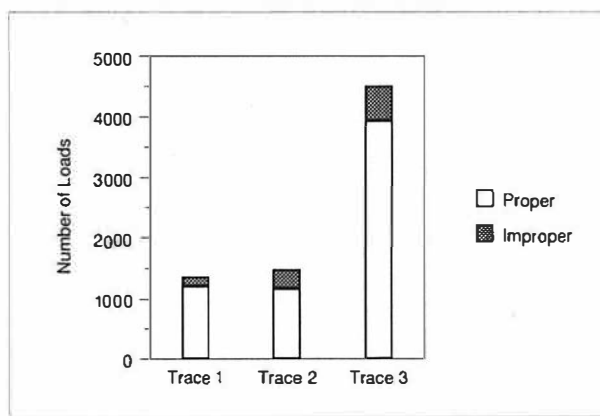


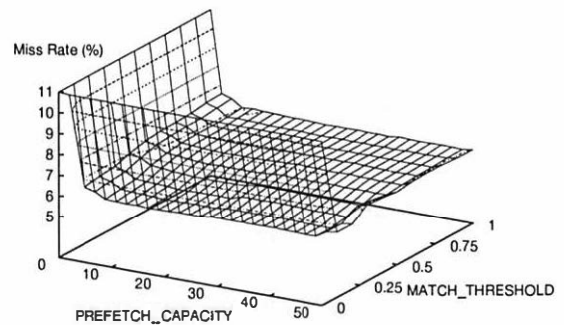
Figure 4: **Effectiveness of the compatibility Function.** This figure illustrates, for each trace, the split of pattern tree loads into proper ones and improper ones.

To illustrate the stability of the algorithmic parameters, we measure the mechanism over a wide range of parameter values. In Figure 5, we show the results for Trace 3 and a cache size of 50. Three measurements are presented: cache miss rate, predication accuracy and prefetch overhead. The results for the other two traces and cache sizes are similar.

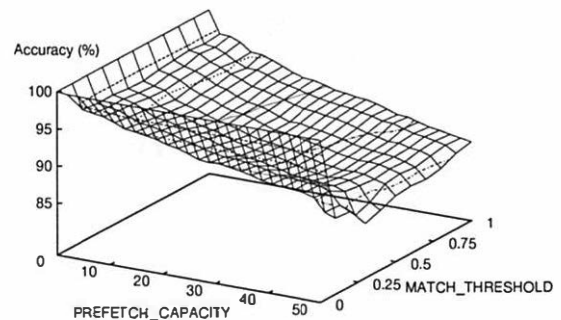
The simulation demonstrates that our algorithm can accurately predict future file accesses based on past file usage. The major limitation of the simulation study is that it does not account for the relative timing of events due to the absence of such information in the traces used. As a result, prefetched files are assumed to appear in the cache instantaneously. It remains to be determined whether a real system will have the resources to exploit the information on future file accesses. The limitations of the simulation motivated us to conduct a full implementation and further evaluations.

## 4 Implementation

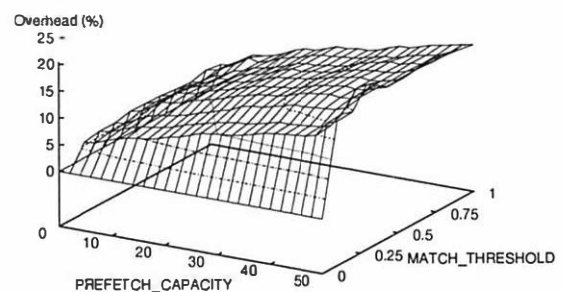
We have implemented our mechanism in UX42 [8], a BSD UNIX server running on Mach 3.0 [1]. UX42 resides in user space and is organized as a collection of C threads [5]. Most threads handle BSD system calls. Among the others are NFS [23] *async daemons*, which handle asynchronous NFS block I/O requests. Since we expect that network file accesses would be the performance bottleneck in a client-server architecture, we prefetch only NFS files opened for read.



(a) Cache Miss Rate



(b) Prediction Accuracy



(c) Prefetch Overhead

Figure 5: **Mechanism Stability.** This figure presents measurements of the prefetching mechanism over a wide range of parameter values. `MATCH_THRESHOLD` was varied from 0 to 1. `PREFETCH_CAPACITY` from 0 (no prefetching) to 50. The measurements were taken for Trace 3 and a cache size of 50.

## 4.1 Structure

Figure 6 shows the basic structure of the implementation, where each box stands for a C thread and the shaded area constitutes the prefetcher. The prefetcher consists of two pieces of code. The first is the system independent prefetch engine, which handles prefetch analysis, working tree construction and pattern tree maintenance. The same code was used in the simulation. In the implementation, the BSD service threads were modified to provide the prefetch engine information on each `fork`, `execve`, `open`, `chdir` and `exit` system call. The prefetch engine processes this information, makes prefetch decisions and enters the files to be prefetched in a queue. The second piece of prefetcher code is an added thread called the prefetch daemon. The prefetch daemon consumes the file queue and produces block read requests in another queue. These requests are then satisfied by the async daemons. Unlike a user-initiated read operation, a prefetch ends when the block is placed in the system buffer cache. No copy to user space is necessary.

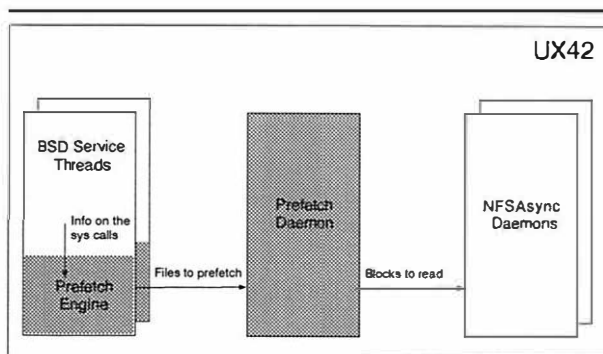


Figure 6: **Structure of Implementation.** A prefetch daemon is added to the collection of C threads in UX42. Each BSD service thread is extended to call the prefetch engine when a relevant system call is serviced.

The prefetch daemon takes advantage of the NFS readahead logic by prefetching only the first block of a file. When a requested block number is one more than the number of the block last read (`r_lastr`), NFS readahead logic speculates that the file is being accessed sequentially, and initiates an asynchronous read on the next block together with the requested block. At least two requested blocks are needed to establish the sequential access pattern before NFS starts readahead. Accordingly, when it removes an entry from the file queue, the prefetch daemon gener-

ates a read request for only the first file block (block 0). It also sets `r_lastr` to -1. Later when block 0 is actually accessed, the request will hit in the cache, and, since `r_lastr` is -1, a readahead on block 1 will be issued. This scheme moves on as the file is read block by block, which is the norm. Since the prefetch daemon issues only one block read for each file to be prefetched, the cost of prefetching is minimal.

We have ensured that prefetch I/O yields to regular NFS asynchronous I/O. All the NFS async daemons can be used towards regular I/O, but only up to a certain number of them towards prefetch I/O. Prefetch I/O will be started only if this limit has not been reached and there is no pending regular I/O. Thus, regular asynchronous I/O's can be serviced promptly and when there are too many of these, the prefetcher will refrain from issuing any prefetch I/O's. This ensures that prefetching halts when the system capacity limit is reached, and does not add extra load to an overload.

The implementation consists of approximately 3380 lines of C code. Of these, 90 lines have been added to the existent UX42 source files, 2280 lines are in separate ".c" files and 1010 lines in ".h" files.

## 4.2 Controlled Experiments

We started the evaluation of the implementation with two simple, controlled experiments. The following questions motivated our experiments:

- How much are the potential benefits of prefetching?
- Under antagonistic circumstances, what is the bearing of prefetching on performance?
- What is the CPU overhead due to prefetching?

The first experiment is a shell script that consists of tens of filter programs, each of which reads one parametric input file, performs some transformation, and writes one output file. Many well known UNIX programs are filters; included in our script are: `awk`, `compress`, `sed`, `sort`, `strings`, `uniq`, `uencode`, and members of the `grep` family. All input files are the same size and reside remotely.

The second experiment is composed of several program builds. Although builds may seem ideally suited to prefetching, they are antagonistic, for two reasons. First, on our hardware platform a build is CPU intensive, leaving the prefetch mechanism relatively little excess CPU cycles with which to make prefetch decisions. Second, compilations often access header files rapid-fire, meaning that the time between a prefetch decision and the actual need for the

| File Size<br>(blocks) | Cache Miss Rate |       |            |       | Latency      |              |         |
|-----------------------|-----------------|-------|------------|-------|--------------|--------------|---------|
|                       | Buffer Cache    |       | Name Cache |       |              |              |         |
|                       | No PF           | PF    | No PF      | PF    | No PF        | PF           | Speedup |
| 1                     | 41.67%          | 0.00% | 68.47%     | 5.87% | 20.34 (0.29) | 12.21 (0.52) | 1.67    |
| 2                     | 52.63%          | 0.00% | 68.47%     | 4.26% | 25.92 (0.17) | 17.43 (0.53) | 1.49    |
| 4                     | 30.30%          | 0.00% | 68.47%     | 4.64% | 34.24 (1.18) | 24.69 (0.56) | 1.39    |
| 8                     | 16.39%          | 0.10% | 68.47%     | 4.45% | 46.39 (0.27) | 37.47 (0.82) | 1.24    |

(a)

| File Size<br>(blocks) | Cache Miss Rate |       |            |       | Latency      |              |         |
|-----------------------|-----------------|-------|------------|-------|--------------|--------------|---------|
|                       | Buffer Cache    |       | Name Cache |       |              |              |         |
|                       | No PF           | PF    | No PF      | PF    | No PF        | PF           | Speedup |
| 1                     | 41.67%          | 1.04% | 68.47%     | 8.14% | 23.94 (0.54) | 14.33 (0.89) | 1.67    |
| 2                     | 52.63%          | 0.88% | 68.47%     | 6.72% | 30.07 (1.24) | 20.34 (0.66) | 1.48    |
| 4                     | 30.30%          | 1.07% | 68.47%     | 6.63% | 38.76 (1.10) | 28.38 (1.24) | 1.37    |
| 8                     | 16.39%          | 0.07% | 68.47%     | 4.92% | 54.04 (1.28) | 42.94 (1.10) | 1.26    |

(b)

Table 3: **Filters Experiment.** This table summarizes the performance results of the filters experiment. Part (a) presents the results with the 10 Mb/sec wired link; part (b) with the 2 Mb/sec wireless link.

data may not be sufficient to complete the prefetch I/O.

The experiments were run standalone. Both the client and the server are 486 processors with 16MB of memory. The client dedicates 1.57MB to its UNIX buffer cache. Since we are interested to find out how our mechanism behaves in relation to different network bandwidth, we ran the tests using both hardwired and wireless links directly connecting the client and server. The hardwired connection is an Ethernet (10 Mb/sec), while the wireless link is an NCR "WaveLAN" [26] radio link with a maximum 2 Mb/sec data rate. For each combination of experiment and network link, we ran the tests both with and without prefetching. Each number reported below is the mean of three trials.

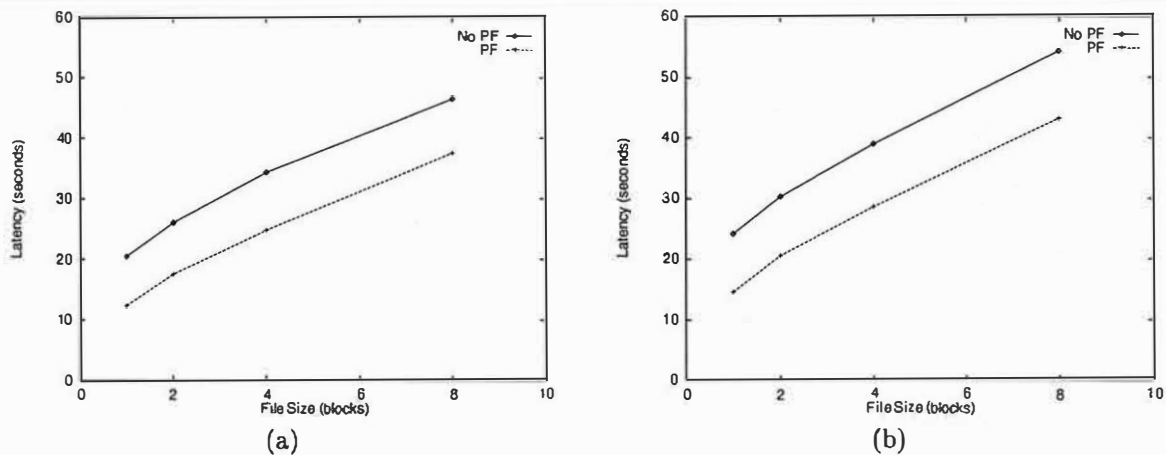
Table 3 shows the results when the filters experiment is run with a wired link. We varied the work load by using different input file sizes, as given in the "File Size" column. Size is measured in multiples of the server's preferred NFS block size, which is 4KB. For each work load, we list the UNIX buffer cache miss rate as well as the directory name lookup cache miss rate. Since our mechanism deals exclusively with network files, the cache miss rates are those of remote entries. The reduction of the miss rates due to prefetching is substantial. We also present the measurements of application latency, or total elapsed time, which we couldn't do in the simulation. From a user's standpoint, latency is the most impor-

tant performance metric. The time measurements are in seconds, with standard deviations included in parentheses. The "Speedup" column is the ratio of no-prefetching latency to prefetching latency. The speedups are significant, particularly when the input files are relatively small, since the delay caused by a sequential file access mainly lies in the first one or two block reads. The remaining blocks, if any, will be brought into cache by NFS readahead logic before they are needed.

Table 3(b) shows the results when the same experiment is run over a wireless link. The application latency is larger because of the slower link, but the speedups are comparable to those in a wired setup. The cache miss rates with prefetching are slightly higher over a wireless link than over a wired link since a small amount of prefetch operations cannot be completed in time for demanded use. Nevertheless, the bandwidth of the wireless link is adequate to perform NFS readahead on a timely basis, so it still suffices for the prefetcher to read only the first file block for each prediction it makes.

Figure 7 graphically illustrates the application latency of the filters experiment.

Our second experiment consists of builds of several UNIX utility programs. Table 4 contains the results of these tests. The application latency is also illustrated in Figure 8. When a large number of header files are opened in quick succession — common during compilation — the prefetcher often does not have



**Figure 7: Filters Experiment: Comparison of Application Latency.** These graphs illustrate the latency data in Table 3. Part (a) compares latency in the wired setting; part (b) in the wireless setting. Prefetching results in significant speedup in this experiment.

enough time or available CPU cycles to make a fruitful prefetch even though it can speculate accurately which files will soon be referenced. The relatively small name cache miss rate improvements confirm this. However, prefetching still manages to reduce the buffer cache miss rate by 65% to 85%. There is no significant latency enhancement, suggesting that the total file read time is not dominant in the application latency. We are pleased to see that no negative effects are observed in an adverse case such as this.

We have used simulation traces to demonstrate the prefetch overhead in terms of wasted network bandwidth and server capacity. The implementation enables us to collect data on another type of cost, extra CPU consumption. Table 5 presents the results for the two experiments. The CPU time includes the system time and user time. Again the time measurements are in seconds and the standard deviations are given in parentheses. The CPU time overhead is negligible in all cases. We also show the ratio of CPU time to application latency ( $CPU/latency$ ). It comes as no surprise that prefetching increases this ratio, substantially in the first experiment. A prefetcher reduces the total elapsed time by increasing the parallelism between CPU processing and I/O.

### 4.3 Discussion

There are three necessary conditions for prefetching to be useful. First, there must be spare capacity in the whole “data pipe” that extends from server’s disk to client’s cache. This pipe consists of

the client’s network I/O interface, the network, the server’s CPU, and the server’s disk and network I/O paths. Second, there must be system resources on the client side for the prefetcher’s use. Of special importance are CPU cycles. The prefetcher cannot fulfill its duty if it doesn’t acquire CPU cycles promptly, even though the amount of CPU time needed is low. Third, the workload must allow some interval between file accesses so that the prefetched I/O can be started and completed ahead of the demand.

Prefetching proves feasible on both wired and wireless network connections. Although our experiment results suggest that the speedups for these two situations are comparable, the speedup may be even better appreciated by the client at the end of a wireless link. With the slower link, a job lasts longer and more real time can be saved via prefetching. As the ratio of CPU speed to network speed increases, prefetching should provide more benefit. However, there is some subtlety with our implementation scheme. When the ratio is lowered to a certain point, we will no longer be able to depend on the NFS readahead function to bring in subsequent file blocks quickly enough. This difficulty can be tackled by a simple generalization of our initial approach. The prefetcher can always prefetch the first  $N$  file blocks ( $N$  growing with the CPU/network ratio) and the NFS readahead code can be modified to read the next  $N$ -th file block, instead of the immediately next block. Currently,  $N$  is simply set to 1.

| Utility  | Cache Miss Rate |       |            |       | Latency      |              |         |
|----------|-----------------|-------|------------|-------|--------------|--------------|---------|
|          | Buffer Cache    |       | Name Cache |       |              |              |         |
|          | No PF           | PF    | No PF      | PF    | No PF        | PF           | Speedup |
| hashinfo | 32.09%          | 5.30% | 8.39%      | 6.92% | 21.59 (0.32) | 19.53 (0.53) | 1.11    |
| snames   | 17.15%          | 2.85% | 9.03%      | 7.84% | 30.93 (0.44) | 29.81 (0.49) | 1.04    |
| machid   | 10.57%          | 3.17% | 10.35%     | 9.83% | 64.11 (1.05) | 63.63 (0.68) | 1.01    |
| machipc  | 23.42%          | 3.36% | 8.48%      | 6.90% | 25.48 (0.78) | 24.70 (2.51) | 1.03    |

(a)

| Utility   | Cache Miss Rate |       |            |       | Latency      |              |         |
|-----------|-----------------|-------|------------|-------|--------------|--------------|---------|
|           | Buffer Cache    |       | Name Cache |       |              |              |         |
|           | No PF           | PF    | No PF      | PF    | No PF        | PF           | Speedup |
| hash.info | 32.09%          | 5.61% | 8.46%      | 6.99% | 26.22 (0.66) | 24.79 (0.47) | 1.06    |
| snames    | 17.20%          | 2.96% | 9.09%      | 7.91% | 38.17 (1.17) | 33.89 (1.64) | 1.13    |
| machid    | 9.15%           | 3.21% | 10.39%     | 9.86% | 77.31 (3.25) | 74.42 (2.13) | 1.04    |
| machipc   | 23.27%          | 4.03% | 8.55%      | 7.02% | 29.10 (0.36) | 27.17 (0.30) | 1.07    |

(b)

Table 4: **Builds Experiment.** This table summarizes the performance results of the builds experiment. Part (a) presents the results with the 10 Mb/sec wired link; part (b) with the 2 Mb/sec wireless link.

## 5 Related Work

Prefetching is an old idea. It has been studied extensively in various areas, including prepagging, prefetching of files and prefetching of database objects. Prepagging has not had a major impact in computer architecture because of the tight time and complexity constraints on paging hardware and software. However, prefetching of files and database objects is a more promising endeavor for two reasons. First, since file and database accesses are less frequent than page accesses, the speed with which the decision to prefetch must be made is not so much of the essence. Secondly, the resource most needed to arrange intelligent prefetching, namely client CPU cycles, is the resource most in excess in distributed systems now and in the likely future.

Some researchers have looked into prefetching blocks *within* files. Sequential readahead [7, 17] is the most primitive and yet successful approach. The work of Kotz and Ellis (see, for example, [14]) focuses on the uses of prefetching to increase I/O bandwidth on MIMD shared-memory architectures. Their prefetching methods are geared to the patterns of many scientific and database applications running on multiprocessors. In contrast to these methods, our work looks for access patterns *across* files.

Some file prefetching methods require that each application inform the operating system of its future demands. This includes the TIP project by Patterson *et al.* [21, 22] and the work of Cao, Felton, Karlin

and Li [3, 4]. These researchers consider the interaction between prefetching and caching more carefully. TIP uses application-disclosed access patterns to dynamically allocate file buffers between the competing demands of prefetching and caching, based on a cost-benefit model. Cao *et al.* allow applications to pass down both prefetching hints and caching hints. They then employ an integrated algorithm for prefetching and caching, which is shown to be theoretically near-optimal. These “informed” approaches possess an advantage over ours in that prefetching is driven not by deductions made after snooping, but rather by certain knowledge provided in advance by higher levels. There is no danger that disastrously incorrect speculative prefetching might trash the cache. On the other hand, these approaches require re-coding applications. Also, the prefetching mechanism must act within the interval between when the higher level learns of the need to do I/O and when it actually initiates I/O; this interval may not always be sufficient to perform the prefetch I/O.

Like ours, a number of other prefetching methods are completely transparent to clients. They use past accesses to predict future accesses. While we seek to build semantic structures, *i.e.*, access trees, that are endowed with application-level meaning, most of the other approaches use a probabilistic method to model the user behavior. In its most general form, a probabilistic method regards file references as a string of events, and uses information about the last

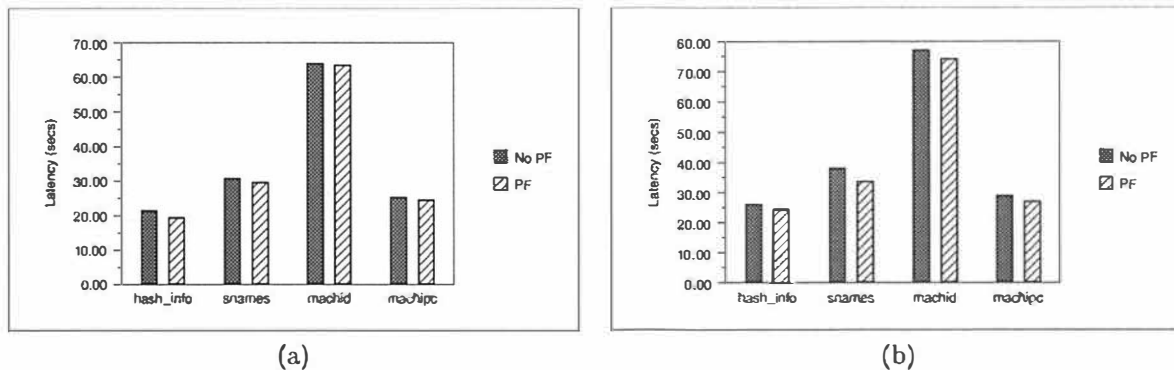


Figure 8: **Builds Experiment: Comparison of Application Latency.** These graphs illustrate the latency data in Table 4. Part (a) compares latency in the wired setting; part (b) in the wireless setting. There are no observable negative effects in this antagonistic experiment.

$C$  events to estimate the frequency distributions of the next  $L$  events ( $C$  and  $L$  are called *context order* and *lookahead size*, respectively). For simplicity, however, existing methods have either single context order or single lookahead. Unlike our tree-based approach, there is no attempt to build in an “understanding” of why files are likely to be referenced together.

Some recent examples of work based on probabilistic methods are a paper by Curewitz *et al.* [6] on prefetching objects in an object-oriented database, work by Griffioen and Appleton [9, 10] and by Kroeger and Long [15] that prefetch whole files. Both [6] and [15] adapt context modeling techniques used in data compression to predict the next access. Their work is inspired by the idea that a good data compressor should be able to predict future data well. Griffioen and Appleton’s work, in comparison, employs a “probability graph” that for each file accumulates frequency counts of all files that are accessed within the lookahead window.

In the initial stages of our work, we considered probabilistic modeling. We implemented (in the simulator only) two extremely simple probabilistic methods, which we called “stupid pairs” and “smart pairs:”

- **Stupid pairs:** When file  $F$  is accessed, prefetch the file that was accessed immediately following  $F$  at the last time  $F$  was accessed.

Sample series of accesses:  $F$ ,  $G$  (remember  $F$ - $G$ ),  $F$  (prefetch  $G$ , remember  $G$ - $F$ ),  $H$  (remember  $F$ - $H$ ),  $F$  (prefetch  $H$ , remember  $H$ - $F$ ),  $H$  (cache hit).

- **Smart pairs:** Keep track of all files that are

accessed immediately after  $F$ , and when  $F$  is accessed the next time, choose one according to the frequency distribution.

Sample series of accesses (only  $F$ ’s pairs are shown):  $F$ ,  $G$  ( $F$ - $G1$ ),  $F$  (prefetch  $G$ ),  $H$  ( $F$ - $[G1, H1]$ ),  $F$  (prefetch  $G$  or  $H$  with 1:1 weighting),  $H$  ( $F$ - $[G1, H2]$ ),  $F$  (prefetch  $G$  or  $H$  with 1:2 weighting).

The stupid pair scheme worked better than the smart pair approach when applied to our traces. This unexpected result is explained by considering the locality of many accesses: often, a user works on one file or one group of files for some time, then moves on to similar operations with different files. Stupid pairs are well-equipped to handle this usage pattern, since they invariably prefetch the most recently used successor file. The seemingly superior intelligence of smart pairs actually becomes a liability when locality is strong; files no longer in active use may be given undue weight if they were heavily accessed in the past.

The phenomenon, that less information is better provided that it is more recent, identifies one common weakness of existing probabilistic methods: they don’t address the problem of aged information. Instead they make predictions based on a total history of accesses. In addition, single lookahead methods like [6] and [15] are less likely to be widely applicable: as I/O latencies grow in terms of CPU cycles, prefetches must begin ever further in advance if they are to complete in time. On the other hand, single context order methods like that of Griffioen and Appleton fail to make full use of historical file access information, and thus are unable to confidently in-

| File Size | Wired Link (10 Mb/sec) |              |             |        | Wireless Link (2 Mb/sec) |              |             |        |
|-----------|------------------------|--------------|-------------|--------|--------------------------|--------------|-------------|--------|
|           | CPU Time               |              | CPU/latency |        | CPU Time                 |              | CPU/latency |        |
|           | No PF                  | PF           | No PF       | PF     | No PF                    | PF           | No PF       | PF     |
| 1         | 3.82 (0.24)            | 4.50 (0.24)  | 18.80%      | 36.87% | 3.69 (0.07)              | 4.37 (0.07)  | 15.41%      | 30.47% |
| 2         | 5.39 (0.71)            | 5.80 (0.48)  | 20.81%      | 33.29% | 4.77 (0.09)              | 5.83 (0.14)  | 15.86%      | 28.65% |
| 4         | 7.88 (0.49)            | 8.78 (0.79)  | 23.00%      | 35.58% | 7.39 (0.22)              | 8.60 (0.35)  | 19.07%      | 30.29% |
| 8         | 13.24 (1.03)           | 14.45 (0.53) | 28.54%      | 38.58% | 13.14 (0.08)             | 14.58 (0.08) | 24.32%      | 33.95% |

(a)

| Utility   | Wired Link (10 Mb/sec) |              |             |        | Wireless Link (2 Mb/sec) |              |             |        |
|-----------|------------------------|--------------|-------------|--------|--------------------------|--------------|-------------|--------|
|           | CPU Time               |              | CPU/latency |        | CPU Time                 |              | CPU/latency |        |
|           | No PF                  | PF           | No PF       | PF     | No PF                    | PF           | No PF       | PF     |
| hash_info | 5.19 (0.21)            | 5.84 (0.37)  | 24.03%      | 29.91% | 5.17 (0.11)              | 5.79 (0.21)  | 19.72%      | 23.36% |
| snames    | 10.36 (0.67)           | 11.25 (0.80) | 33.48%      | 37.73% | 10.69 (0.10)             | 11.16 (0.40) | 28.00%      | 32.94% |
| machid    | 30.29 (0.92)           | 32.08 (1.28) | 47.24%      | 50.41% | 30.96 (1.82)             | 32.58 (1.70) | 40.04%      | 43.78% |
| machipc   | 6.74 (0.22)            | 7.63 (0.45)  | 26.45%      | 30.90% | 6.91 (0.27)              | 7.80 (0.34)  | 23.73%      | 28.72% |

(b)

**Table 5: CPU Time Consumption.** This table presents the CPU time with and without prefetching, over two different network links. Also given is the ratio of CPU time to application latency. Part (a) give the results for the filters experiment; part (b) for the builds experiment.

fer accesses far into the future.

Similar to our work, Palmer and Zdonik's work on Fido [20] also explicitly recognizes and maintains access patterns. Several important aspects make their work different. First, their work was conducted in the context of object-oriented database systems, whereas our context is file systems. Second, they represent access patterns with strings of object identifiers, with no semantics involved. We represent patterns with access trees. Third, they employ specialized pattern memory, while we store pattern trees in virtual memory. Finally and most notably, Fido requires a separate training phase after each user session, while our mechanism is more on-line: there is no off-line computation or periodic analysis needed.

An issue that is related to prefetching is hoarding [12, 11, 16, 25]. Both prefetching and hoarding involve anticipatory file fetches: bringing files from remote servers into a local cache before they are needed. These are not exactly the same techniques, however. Hoarding is a scheme designed to increase the likelihood that a mobile client will be able to continue working during periods of total disconnection from file servers. Since hoarding is a relatively infrequent operation performed only at a client's request prior to disconnection, timing is not critical. On the other hand, prefetching is mainly concerned with improving performance and timing is impor-

tant. With prefetching, the file server is assumed to be still accessible, although the network connectivity may be weak. A cache miss is much more catastrophic in disconnected operations, hence hoarding is typically willing to overfetch substantially in order to enhance the availability of files. Despite the differences, our idea of uncovering and exploiting the semantic structure underlying file references also applies to the hoarding problem, as shown in [25].

## 6 Conclusion

We have presented a technique for transparent on-line file prefetching. The technique analytically models interesting system calls and builds semantic structures that capture the intrinsic correlations between file references. It makes accurate predictions of future file accesses, imposes little CPU overhead, defers to demand I/O, and delivers substantially lower client cache miss rates and elapsed time for I/O-intensive applications.

One central trait of the algorithm is that it spends client CPU cycles in return for more effective use of client cache space and fewer on-demand network operations. Another distinguishing aspect is that the algorithm's lookahead ability is potentially much greater than that of previous work. Both of these traits help to couple application I/O performance more closely to CPU speed than to I/O device speed,

thereby addressing a fundamental and longstanding problem in operating systems [18].

Our initial performance evaluation has been encouraging. We intend to extend the evaluation along several directions. First, we would like to run experiments that model user behavior more realistically. We are in the process of synthesizing workloads based on actual file system traces. Second, we plan to conduct experiments over a much wider spectrum of network and client capacities. We hope to reach a good understanding on when prefetching is feasible. Third, we would like to examine the applicability of our prefetching algorithm to different kinds of file systems: local file systems, remote file systems with caching of file blocks in main memory only, and remote file systems with whole file caching on client disks.

## 7 Acknowledgements

This work was supported in part by the Advanced Research Projects Agency, ARPA order number B094, under contract N00014-94-1-0719, monitored by the Office of Naval Research; and in part by the Center for Telecommunications Research, an NSF Engineering Research Center supported by grant number ECD-88-11111.

We wish to thank Carl Tait for many constructive conversations and insightful comments. We also wish to thank the anonymous reviewers and our USENIX shepherd, Carl Staelin, for their remarks.

## References

- [1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. In *Proc. 1986 USENIX Summer Conf.*, pages 93–112, June 1986.
- [2] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a Distributed File System. In *Proc. Thirteenth Symp. on Operating System Principles*, pages 198–212. ACM, October 1991.
- [3] P. Cao and E. W. Felten and A. Karlin and K. Li. A Study of Integrated Prefetching and Caching Strategies. In *Proc. 1995 ACM SIGMETRICS*, pages 171–182, June 1995.
- [4] P. Cao and E. W. Felten and A. Karlin and K. Li. Implementation and Performance of Integrated Application-Controlled Caching, Prefetching and Disk Scheduling. In *Proc. First USENIX Symposium on Operating Systems Design and Implementation*, pages 165–178, November 1994.
- [5] E. C. Cooper and R. P. Draves. C Threads. Technical Report CMU-CS-88-154, Carnegie Mellon University, June 1988.
- [6] K. M. Curewitz, P. Krishnan, and J. S. Vitter. Practical Prefetching via Data Compression. In *Proc. 1993 ACM SIGMOD*, pages 257–266, May 1993.
- [7] R. J. Feiertag and E. I. Organisk. The Multics Input/Output System. In *Proc. Third Symp. on Operating System Principles*, pages 35–41. ACM, 1971.
- [8] D. Golub, R. Dean, A. Forin, and R. Rashid. Unix as an Application Program. In *Proc. Summer 1990 USENIX Conf.*, USENIX, pages 87–95, June 1990.
- [9] J. Griffioen and R. Appleton. Reducing File System Latency Using a Predictive Approach. In *Proc. 1994 USENIX Summer Conf.*, pages 197–207, June 1994.
- [10] J. Griffioen and R. Appleton. Performance Measurements of Automatic Prefetching. In *Parallel and Distributed Computing Systems*, pages 165–170, IEEE, September 1995.
- [11] L. B. Huston and P. Honeyman. Disconnected Operation for AFS. In *Proc. First USENIX Symp. on Mobile and Location-Independent Computing*, pages 1–10, August 1993.
- [12] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. In *Proc. Thirteenth Symp. on Operating System Principles*, pages 213–225. ACM, October 1991.
- [13] K. Korner. Intelligent Caching for Remote File Service. In *Proc. Tenth Intl. Conf. on Distributed Computing Systems*, pages 220–226. IEEE, May 1990.
- [14] D. Kotz and C. S. Ellis. Practical Prefetching Techniques for Parallel File Systems. In *Proc. First Intl. Conf. on Parallel and Distributed Information Systems*, pages 182–189. ACM, December 1991.
- [15] T. M. Kroeger and D. D. E. Long. Predicting Future File-System Actions from Prior Events. In *Proc. 1996 USENIX Annual Technical Conf.*, pages 319–328, January 1996.

- [16] G. H. Kuenning. The Design of the Seer Predictive Caching System. in *Proc. Workshop on Mobile Computing Systems and Applications*, ACM/IEEE, pages 37–43, December 1994.
- [17] M. K. McKusick, W. J. Joy, S. J. Leffler, and R. S. Fabry. A Fast File System for Unix. *ACM Trans. on Computer Systems*, 2(3):181–197, August 1984.
- [18] J. K. Ousterhout. Why Aren't Operating Systems Getting Faster As Fast As Hardware? In *Proc. 1990 USENIX Summer Conf.*, pages 247–256, June 1990.
- [19] J. K. Ousterhout, H. Da Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *Proc. Tenth Symp. on Operating System Principles*, pages 15–24. ACM, December 1985.
- [20] M. Palmer and S. Zdonik. Fido: A Cache That Learns to Fetch. In *Proc. 17th Intl. Conf. on Very Large Data Bases*, pages 255–264, September 1991.
- [21] R. H. Patterson, G. A. Gibson, and M. Satyanarayanan. A Status Report on Research in Transparent Informed Prefetching. *Operating Systems Review*, 27(2):21–34, April 1993.
- [22] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky and J. Zelenka. Informed Prefetching and Caching. In *Proc. Fifteenth Symp. on Operating System Principles*, pages 79–95. ACM, December 1995.
- [23] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network Filesystem. In *Proc. 1985 USENIX Summer Conf.*, pages 119–130, June 1985.
- [24] C. D. Tait and D. Duchamp. Detection and Exploitation of File Working Sets. In *Proc. Eleventh Intl. Conf. on Distributed Computing Systems*, pages 2–9. IEEE, May 1991.
- [25] C. D. Tait, H. Lei, S. Acharya and H. Chang. Intelligent File Hoarding for Mobile Computers. In *Proc. First Intl. Conf. on Mobile Computing and Networking*, pages 119–125, ACM, November 1995.
- [26] B. Tuch. An Engineer's Story of WaveLAN. In *Proc. First Virginia Tech Symp. on Wireless Personal Communications*, June 1991.

# Optimistic Deltas for WWW Latency Reduction

Gaurav Banga

*Rice University*

gaurav@cs.rice.edu, <http://www.cs.rice.edu/~gaurav/>

Fred Douglass

*AT&T Labs – Research*

douglass@research.att.com, <http://www.research.att.com/~douglass/>

Michael Rabinovich

*AT&T Labs – Research*

misha@research.att.com, <http://www.research.att.com/~misha/>

## Abstract

When a machine is connected to the Internet via a slow network, such as a 28.8 Kbps modem, the cumulative latency to communicate over the Internet to World Wide Web servers and then transfer documents over the slow network can be significant. We have built a system that optimistically transfers data that may be out of date, then sends either a subsequent confirmation that the data is current or a delta to change the older version to the current one. In addition, if both sides of the slow link already store the same older version, just the delta need be transferred to update it.

Our mechanism is optimistic because it assumes that much of the time there will be sufficient idle time to transfer most or all of the older version before the newer version is available, and because it assumes that the changes between the two versions will be small relative to the actual document. Timings of retrievals of random URLs in the Internet support the former assumption, while experiments using a version repository of Web documents bear out the latter one. Performance measurements of the optimistic delta system demonstrate that deltas significantly reduce latency when both sides cache the old version, and optimistic deltas can reduce latency, to a lesser degree, when content-provider service times are in the range of seconds or longer.

## 1 Introduction

The Internet, and particularly the World Wide Web ( $W^3$ ), consists of an ever-increasing number of servers, networks, and personal machines with dra-

matically varying qualities of service. Individuals often access the  $W^3$  via modems with bandwidth of 14.4–28.8 Kbps, while their provider might have a T1 link or better to the rest of the Internet. But then the actual access may be to another site with low bandwidth, high server load, or both. Thus the latency to respond to the user's request for a page is unpredictable: often fairly low, but sometimes extremely high. The unpredictability and generally slow response may be exacerbated in environments with even poorer quality of service, such as cellular telephone or wide-area wireless networks.

A number of techniques have been implemented or proposed to deal with HTTP latency. A browser can direct its request to a proxy-caching server (henceforth referred to as a *server proxy*) on the other end of the low-speed connection, and then the latency for retrieving pages elsewhere in the Internet can be eliminated when someone else has retrieved those pages in the recent past. (Recency is a function of the size of the cache, any expiration dates in the pages, and any constraints passed from the browser to the cache [5, 12]. Also, some pages are flagged as uncacheable, and the proxy-caching server is obliged to pass those requests through to the content provider. Caching is discussed further in Section 2.) America Online (AOL) uses a proprietary protocol between the browser on a user's machine and an enormous cache of  $W^3$  pages within the AOL server cluster. Prefetching pages during periods when the modem would otherwise be idle can reduce or eliminate the latency of following a link, if the prefetch is accurate and the user thinks between clicks long enough for prefetching to complete [18, 22]. Padmanabhan and Mogul's study of persistent HTTP con-

nections [17] indicates that a persistent connection between a client and proxy, or between one of them and a content provider, will eliminate TCP connection setup and slow-start overhead.

We look at the problem of latency from another perspective: using computation to improve end-to-end network latency. The idea of trading off computation for I/O bandwidth has appeared numerous times in past systems. Examples include application-specific deltas and compression, such as Low-bandwidth X[15]; compressed network or disk I/O [3, 6, 7]; replicated file systems [2]; shared memory [16]; and checkpointing [9, 19]. It seems that the same tradeoffs apply in the domain of the  $W^3$ .

We address the issue of latency from the perspective of sending the differences between versions of a page, or *deltas*, in order to avoid sending entire pages. Briefly, deltas are used in two ways. First, if both ends of the slow link store the same version of a page, and the server proxy obtains a new copy of the page, it can send a delta to the user's machine. This hopefully will reduce the transfer time on the slow link. Second, if the user's machine does not store the older version, the server proxy can send a potentially obsolete version of the page immediately and request the new version in parallel. It follows this with a delta against the obsolete version if necessary. Thus, the idle time on the slow link when the server proxy is waiting for a response from the end server is not wasted.

The size of the delta may turn out to be large, in which case the server proxy may have to abort sending the stale version (if one is being sent) and just send the current page received. In this case, work for sending the stale data and calculating the delta is wasted. Worse, as the server proxy does not pipeline the data from the content provider to the client but instead waits till the whole page has been received (since it needs to compute a delta), the end latency as perceived by the client may be somewhat larger. Our approach optimistically assumes that this case is uncommon; hence we refer to the case where stale data is transferred as an *optimistic delta*. The experiments described in this paper support this assumption. In contrast, we refer to the case where both sides share a cached version as a *simple delta*.

As we were going to press, we found that the "simple deltas" case is similar to a system from IBM called WebExpress [13]. WebExpress is geared toward a low-bandwidth wireless environment, where bandwidth is precious, so it has similar goals but makes different trade-offs. It focusses on small changes to dynamic data (CGI output), sending deltas between a base version that is shared by the client- and server-side "intercepts" (similar to the client and server proxies de-

scribed here). However, WebExpress has apparently not been used so far for arbitrary  $W^3$  pages, nor does it send stale data optimistically. The limited bandwidth, high error rate, and contention in a wireless environment suggest that transferring data that may not be used could have more negative consequences than over a single-user modem.

Our work also has some similarity to Dingle and Partl [5], who proposed that a hierarchy of proxy-caching servers could be used to send stale data as a MIME multipart document, causing the browser to display the stale data immediately and to replace it with more recent versions as they become available. The main difference with our work is that Dingle and Partl do not address the latency of obtaining the final version of the data. Our approach attempts to reduce this latency by sending the the current version as a delta. In addition, we do not display the stale page: it is intercepted by a *client proxy* [20], typically co-located with the browser on the same machine, and passed to the browser once it is updated or known to be current. In the case where the "stale" page is actually current, our system behaves like the proposal in [5]. Finally, intercepting the stale data by a client-side proxy permits the transmission of the stale data to be aborted transparently once it becomes clear that simply sending the current version is more efficient (e.g., if the delta turns out to be too large relative to the page, or if the current page became available very quickly).

The rest of this paper is organized as follows. Section 2 provides some background into caching in the HTTP protocol and an analysis of HTTP latency. Section 3 discusses some data analysis to support our hypotheses that delta sizes will be small and that there will be sufficient idle time to transfer stale copies. Next, Section 4 describes the design of our system, and Section 5 covers experimental results. Finally, Section 6 discusses the status of the system and future work, and Section 7 offers some conclusions.

## 2 Background

In this section we briefly describe caching in HTTP and analyze the dynamics of a typical HTTP transfer. This description provides background for the discussion in the following sections.

### 2.1 Caching in the HTTP Protocol

The HyperText Transfer Protocol (HTTP) [4] supports caching in clients (i.e. browsers) and intermediate servers known as proxy-caching servers. To display a page, a client without a cached copy will unconditionally send a page request to the content-provider or a

proxy-caching server. A client with the page already cached may return the cached copy or contact another host to determine whether the page has changed. This check for page currency is done by sending an HTTP GET request with a header specifying If-Modified-Since followed by the timestamp of the cached copy. If a proxy-caching server is used, it can respond to a page request or currency check request if it has a cached copy that is deemed usable and if the client has not specified that the cache must not be used (with a `Pragma: no-cache`<sup>1</sup> directive, most commonly sent when a user tells a browser to reload a fresh copy of a page).

The proxy-caching server decides whether or not the cached page is usable and, if so, whether or not the client's cached copy is current, based on the optional extra information that the clients can send with their requests and which HTTP servers can send back with a page. Beyond the `Pragma: no-cache` directive mentioned above, the client may specify a bound on the age of the cached copy it is willing to accept (the `Cache-control: max-age` directive). Content-providers can specify when the page was last modified, whether or not the page can be cached (the `Cache-control: no-cache` field), and how long a client or proxy-caching server should cache the page (the `Expires` field). Dynamic data, often the output of a CGI script, is typically sent with no Last-Modified timestamp and set up to expire from the cache immediately (equivalent to disabling caching).

Currently, if a page has no Last-Modified timestamp, checking for the freshness of a cached copy requires retrieving the fresh copy from the content provider and shipping it all the way to the client browser. Similarly, changes to a page with the timestamp will require the proxy to obtain the file from the content provider and transmitting the entire file to the client; the transmission is elided only if the page has not been modified at all.

## 2.2 Analysis of HTTP Latency

We now present an analysis of the timing dynamics of a typical HTTP transaction. We assume a setting where the Web browser (client) talks to a server proxy on the other side of a low bandwidth and high latency link, which in turn talks to HTTP servers (content-providers) on the Internet. We use this analysis to make a case for the usefulness of optimistic deltas.

Consider Figure 1, which depicts the timeline corresponding to an HTTP transaction between a client, a

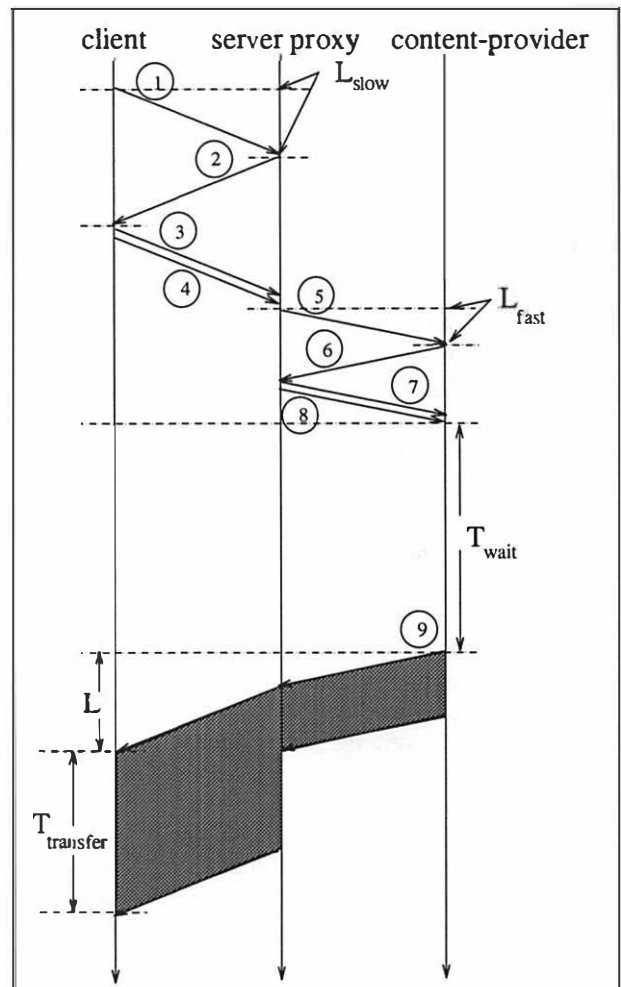


Figure 1: Timeline for typical HTTP request.

server proxy, and a content provider. Packets 1, 2 and 3 correspond to the SYN, SYN/ACK and ACK packets that are exchanged as part of the TCP 3-way handshake in the connection establishment phase between the client and the server proxy. The client sees this connection as established after a delay of approximately  $2 * L_{slow}$  where  $L_{slow}$  is one half of the round-trip latency of the link between the client and the server proxy. At this point the client sends packet 4 which contains the HTTP request. After the server proxy gets this packet (at approximately time  $3 * L_{slow}$ ), if it needs to go to the end server to satisfy this request or to validate its cached copy, it initiates a connection with the content provider (packets 5, 6 and 7). The latter connection is established after a further delay of  $2 * L_{fast}$ , where  $L_{fast}$  is one half of the round-trip latency of the link between the server proxy and the content provider.

<sup>1</sup>HTTP 1.1 alternatively supports a `Cache-control: no-cache` directive, which when sent by the client is equivalent to `Pragma: no-cache`. In this paper we refer to the pragma to mean either header.

The server proxy then forwards the client's HTTP request to the content provider (packet 8).

On receipt of this HTTP request the content provider does whatever is necessary to produce the response. This may include several relatively slow activities, such as forking off a child server and/or passing descriptors to it, forking off a script to produce the response document, invoking and waiting for the response from a search engine, etc. For our purposes we will view these activities as a certain latency shown as time  $T_{wait}$  in the figure. The magnitude of  $T_{wait}$  is thus a function of the server in question, the particular URL being served and the load on the server. This time can be highly variable. Some simple experiments on the Internet indicate that it varies from a couple of hundred milliseconds to several seconds. After the response (packet 9) is available and sent off it takes another  $L = L_{slow} + L_{fast}$  time before the client starts seeing it. The amount of time  $T_{transfer}$  that it takes to transfer the response is dependent on the size of the response and speed of the link.

If we put in typical values for the various time parameters in the figure ( $L_{slow} = 160\text{ms}$ ,  $L_{fast} = 60\text{ms}$ ,  $T_{wait}$  varying from 200ms to several seconds,  $T_{transfer}$  for a 2-KByte response at 20 Kbps is  $800\text{ms}^2$ ), we notice that if  $T_{wait}$  is large, the low bandwidth link is idle for a substantial portion of the time. Also, regardless of the magnitude of  $T_{wait}$  we want to minimize the amount of data we transfer over the slow link. These two observations lead to the idea of optimistic deltas. We want to effectively utilize the slow link in its idle time, and if possible, reduce the amount of data transferred, in order to reduce the total latency perceived by the client. If the client and server proxy both cache the same older version, only a delta needs to be sent over the slow link decreasing  $T_{transfer}$ . If the client and server proxy do not cache the same version, the server proxy can transfer its cached version while waiting for data from the content-provider and subsequently send a delta. Here again,  $T_{transfer}$  is shorter if  $T_{wait}$  is long enough.

### 3 Data Analysis

Simple deltas benefit by trading off computation of the deltas for a reduction in bandwidth and latency over the slow link when both sides store the same old version of a page. Optimistic deltas trade off an *increase* in the amount of data transferred, by sending an older version during an idle time of the slow link followed by a delta, for a reduction in end-to-end latency. The viability of

either form of deltas is thus dependent on "smallness" of deltas, which we evaluate in Section 3.1. Optimistic deltas depend additionally upon long idle times on the slow link, which we consider in Section 3.2.

#### 3.1 Delta Sizes

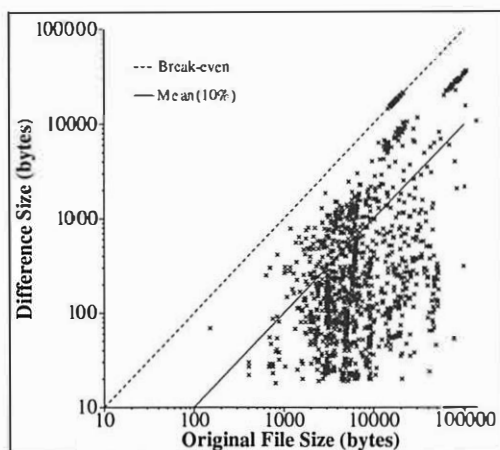
To test the hypothesis that deltas would be sufficiently small, we wanted to take a sample of  $W^3$  pages and see how large the differences were between two versions of the same page, relative to the page itself. We considered two sources of sample pages: the version archive of the AT&T Internet Difference Engine (AIDE) [8], which stores versions of pages for future visual comparison of their changes, and a set of random URLs obtained from AltaVista [1]. The random pages were tracked by AIDE as well, but they were considered separately from those pages that were actually registered explicitly, either by individuals or by inclusion in a list of popular URLs collected from a set of bookmark files within AT&T. The random URLs and popular URLs were archived daily if changes were detected, while the pages tracked for individual users were typically archived upon explicit request.

Throughout our experiments, we computed deltas using *vdelta*, a program that generates compact deltas by essentially compressing the deltas in the process of computing them, and which can be used as a stand-alone compression program as well [10].<sup>3</sup> We must consider the possibility that  $W^3$  pages that are compressed in a stand-alone fashion will compress so well that the deltas between two versions of a page are not much smaller than the compressed page. In this case the client and server proxies could merely compress every page (or rely on compression in the modems) without using deltas and have the same benefit. We will see that in practice, however, deltas are substantially smaller than simple compression.

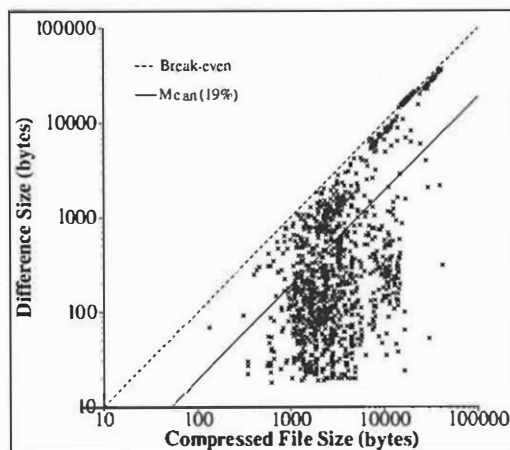
Considering first the non-random pages, of a total of 380 pages in the archive, 181 had more than one version, with a mean of 4.9 versions/page ( $\sigma = 10.3$ ). Figure 2(a) shows a plot of delta size against original file size. The delta size is usually a small fraction of the original file size. By comparison, Figure 2(b) plots delta size against the size of the newer file once compressed. Figure 2(b) indicates that the delta is consistently much smaller than the compressed file, though in some cases it is approximately the same; this usually happens when a file has changed completely from one version to the next. Even if the file does not compress

<sup>2</sup>This number is very approximate and will in practice be larger because of TCP's congestion control algorithms.

<sup>3</sup>In fact, one can consider a delta of  $B$  compared to  $A$  as compressing  $B$  with the strings of  $A$  already in the compressor's table of prefixes: if  $B$  is similar to  $A$  then it will compress well, and if not, it will still compress well if it has internal similarity (as most ASCII text does).



(a) Deltas compared to original file sizes.



(b) Deltas compared to compressed output sizes.

Figure 2: Comparison of sizes of deltas and original or compressed pages, using *vdelta*. While deltas have a greater benefit when simple compression is not taken into account, they help above and beyond the benefits of compression. In each graph, the dashed line indicates the break-even point and the solid line depicts the mean across all files.

well (for instance, it is a GIF file), the worst that *vdelta* will do is to reproduce the original file with a few bytes of overhead. The mean across over 2200 comparisons of delta/compressed-file ratios was 19% ( $\sigma = 27.6\%$ ).

The outlying points in Figure 2, which are due to one GIF file that has been archived automatically each day and a compressed postscript file with two versions in the archive, might be a concern in practice if the system were to send stale copies and compute deltas regardless of file type. Fortunately, file types are identifiable, both from the Content-type HTTP header and data within the files, so it is possible to treat images and other non-textual data specially. One might instead use a distillation technique to send a version of an image that is more appropriate for a low-bandwidth link [11].

Our study of 1000 random URLs from AltaVista [1] found that 861 URLs were actually accessible at the time we started tracking them, and the vast majority (79%) of those URLs were not modified in the next two months of daily checks. Figure 3 graphs the distribution of the number of versions detected for the remaining 21% that were modified. Just 43 of the 861 URLs (5%) had 40 or more versions over the two months of the study; the minor variations in the number of versions of the frequently-changed pages may be due to transient errors while contacting those hosts. We also performed the above analysis of delta sizes for these pages, and found that the mean delta size was just 3.7% of the original page size ( $\sigma = 6.9\%$ ), and 10.4% of

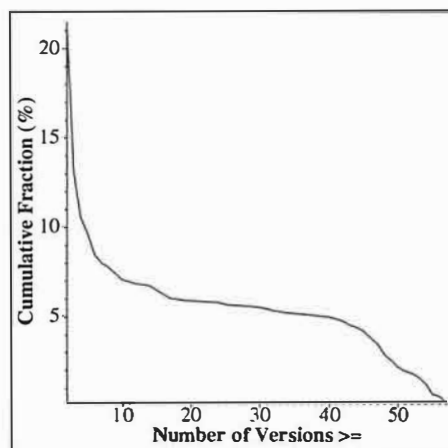


Figure 3: Distribution of the number of versions detected by daily checks of 861 randomly selected URLs over a two-month period, for the 21% of pages that were modified.

the compressed page size ( $\sigma = 14.0\%$ ). The pages themselves compressed to an average of 12.1% ( $\sigma = 18.0\%$ ) of their original size. A possible explanation for the small deltas is that the sample was dominated by pages that changed daily, and those changes may often have been the inclusion of timestamps or other small, simple modifications.

Another consideration is what sorts of data can be compared. Even dynamic pages, which aren't cacheable, might have a lot of overlap between versions of the same page, or pages with the same base URL but different parameters to a CGI script. (Determining when to compare one URL against a slightly different URL for differencing is an open question, but as long as both the client and server proxies agree on the versions being compared the system will act correctly.)

For example, a query to the AltaVista search engine [1] might result in a page containing several links to content and several more links to other URLs within AltaVista. The "boilerplate" can dominate the content that changes from page to page, because each page contains the same form at the top and, at the bottom, a set of links to each other page generated by the query. Figure 4 graphs the sizes of deltas from two queries, compared to the size of the page if it were just compressed. The first, a name lookup, returned 9 pages; the second, a query with many terms ("storage management mobile computing flash memory nvram") that generated thousands of matches, returned 20 pages, of which 10 were compared. In each case the deltas from one page to the next, within a given search result, were much smaller even than the compressed pages.

### 3.2 HTTP Latency

To get a sense for the likelihood that a request would take a long time to start receiving data, we collected 1000 random URLs from AltaVista [1] and timed their responses. This study differs somewhat from Viles and French [21], who studied the availability of random HTTP servers and the time to *connect* to them; here we are seeing how long it takes to collect the first data from a  $W^3$  page. Figure 5 shows the results of this experiment, based on the 722 URLs that returned data within the first minute. We found that about a third of pages responded within a second, assuming they responded at all, and half responded within about 1.6s. However, it takes 5s to cover  $\frac{3}{4}$  of the pages and 10% took 10–30s or more for the first data to arrive. As more pages on the  $W^3$  are dynamically generated, we expect the fraction of pages with sluggish response to increase.

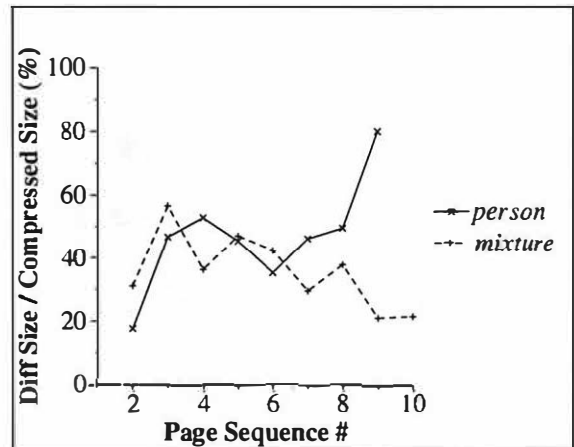


Figure 4: Example of deltas for two AltaVista queries, one for a person and one a mixture of computer science terms. All comparisons were pairwise in sequential order, starting with a delta between the first two pages of a query result. The URL of each page varied slightly because it specified the range of responses to return (1-10, 11-20, etc.).

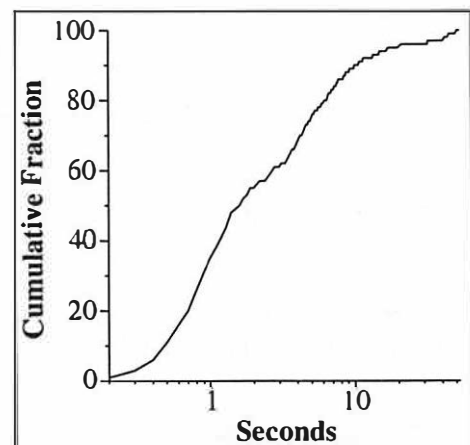


Figure 5: Distribution of response times to receive first data, based on a sampling of 722 URLs. Roughly  $\frac{1}{3}$  of the sample received data within a second, and  $\frac{3}{4}$  received data within 5 seconds.

## 4 System Design

### 4.1 Design Considerations

Once we were confident that sending deltas could often reduce bandwidth requirements and/or client-observed latency of Web access, we had to consider two issues. One was what the architecture of a delta-based system would be. We rejected any scheme that would require changes to HTTP or to existing content providers, though we later learned that HTTP/1.1 will support a PATCH directive to allow efficient uploading of changes to shared documents. Instead, we settled on a proxy-based architecture in which the browser connects to a proxy on the same machine (the client proxy), and that proxy in turn connects to a server proxy on a well-connected host. We have control over both of these proxies; in fact we use a common source code base for them, though that is not necessary. The design of the proxies is covered in greater detail in Section 4.2.

The second open issue was how the deltas would help when the user's machine does not store an old version of the page. If the well-connected proxy has many clients, or can talk to a nearby proxy that does, it may have fast access to a cached copy of the requested page. When that page is *current*, nothing need be done other than sending the cached copy over the phone line. If it is stale, however, there might be an opportunity to use deltas after sending the stale data.

In fact, one might expect the server proxy to handle a number of clients and to keep multiple versions of each document in its cache. Sometimes a client's request will specify a version which the caching proxy has, and then only a delta needs to go back over the slow link. Other times, the server proxy will not have the same version cached as advertised by the client in which case it will try to utilize the  $T_{wait}$  time by sending in the stale copy it has and subsequently sending the delta. Since in the latter scenario the benefit is potentially lower, especially when  $T_{wait}$  is not very large, we may bias the system so that it hits the first scenario more often than the second. This may be brought about by auxiliary mechanisms like prefetching during idle times to keep the client and server proxies' caches in sync.

In total, there are numerous ways in which the optimistic delta mechanism can improve the efficiency of  $W^3$  access:

- The client and server proxies may share an out-of-date version. Consider the case when the client proxy sends an If-modified-since request with a No-cache directive to the proxy, which caches the same version, and assume that the page has been modified on the content provider site. In this case,

the proxy obtains the page, computes the delta and, if it is smaller than the whole page, sends the delta instead of the whole page to the client. Thus, the demand for bandwidth of the (slow) link to the client is decreased. Again, we refer to this case as a "simple delta," which is less "optimistic" than others: it only relies on the delta being small enough to be beneficial but does not risk transferring useless data.

- The server proxy may have the current version, but the client proxy wants to check the validity of its own cached copy with the content provider. Assume the client sends an If-modified-since request with No-cache to the proxy server, which caches a newer version that is the same as the version on the content provider site. In this case, the proxy immediately sends its copy to the client (marked as Stale); in parallel, it sends an If-modified-since request to the content provider, verifies that its copy is actually current and sends a null delta to the client. The browser can display the page as soon as the conditional GET returns via the server proxy, rather than having the newer contents of the page transferred starting then.

The same latency reduction applies if the client has no cached copy but requests the most current version of a page, since the server proxy can send a Stale copy and then confirm that the copy is current after its own If-modified-since request.

- The server proxy may have a newer version than the client, as well as the client's cached version. Assume the client proxy asks the server proxy for a page that the client has cached, and the server's copy is more recent but is not necessarily the most current version. The server can respond with a delta against the client's version. If the page is out of date or the client specifies that the cache be bypassed, the content provider is consulted and a second delta can be sent if needed.
- The client and server proxies may share a current version of an "uncacheable" page, one that must be retrieved directly from the content provider on each access. Our system permits the client and server proxies to cache such pages as a basis for deltas between versions of a page, while ensuring correctness by providing the browser with the most current version every time. The server proxy can determine that the page is unchanged (using a regular GET over the high-speed network and comparing the contents with the cached version) and notify the client proxy to use the cached version rather than transferring the page over the low-

speed network. Note that while other systems do this for cacheable pages, ours does this even for uncacheable ones as far as the slow link is concerned, while taking advantage of deltas when the differences are small.

## 4.2 Architecture

Figure 6 shows the architecture of our system. The browser connects to a local (client) proxy using the usual HTTP proxy-caching mechanism, by sending it requests containing the full URL of a desired page. We assume that the browser does not cache pages, but instead relies on the client proxy; however, this does not affect correctness, only storage utilization and caching effectiveness. The client proxy serves the request out of its cache if possible, or forwards it to the server proxy on the other side of the slow connection. The added overhead of a client proxy on the same machine is minimal by comparison to network delays and so the analysis in Section 2.2 still holds.

The server proxy can respond using its cache if the request is for a cached page, the page is not out of date with respect to its expiration date (if any), and the client has not issued the `no-cache` pragma. Otherwise, it forwards the request upstream, either directly to the content provider or to yet another proxy-caching server. In either case, if the client and server proxies share a cached version, a delta from that version to the current version can be sent once the current version is available. If they do not share a cached version but the server proxy has some version cached, the server can send the possibly stale version followed by a delta from that version to the current version.

The server proxy determines what the client proxy has cached via some extra headers in the HTTP request. First, an `Accepts: multipart/attdelta` field indicates that the client understands the delta format. This way, a browser or other unmodified client can talk to our proxy without getting back something it cannot interpret. Second, a `Current-version: [signature]` field informs the server which version the client has cached, if any. The signature can in principle be anything that can distinguish different versions of a document, such as an MD5 checksum. We make the simplifying assumption that any client proxy that requests a new version from a server proxy will have received the previous version from the same proxy, and we use a monotonically increasing version number (instead of a checksum) that the server generates and passes to its clients.

Table 1 summarizes the possible combinations of client/server proxy states and the procedures that are followed.

## 4.3 Detecting Non-cost-effective Transfers

As was mentioned in the introduction, one difference between our system and the proposal of Dingle and Partl [5] is the ability to abort the transfer of stale data. In fact, there are two cases when it is more appropriate to behave like standard proxies and just send the current version of a page to the client without delta processing.

In the “simple delta” case, the client and server proxies cache the same stale version of the page, and only the delta need be sent. If the delta is as large as the current page, the current page rather than the delta must be sent. If the delta is somewhat smaller, one could use heuristics to decide whether the cost of recreating the current version from the delta exceeds the benefits due to bandwidth reduction. We currently transfer the delta any time it is smaller than the original.

The other case occurs when an optimistic transfer is in progress and the new version of the page starts to arrive at the server proxy. If most of the stale version has been sent and the delta is small, it pays to finish sending the stale version; if there is a lot of data yet to be transferred and/or the delta is large, the transfer should be aborted and the current version should be sent as it becomes available.

If the cost of recreating the page from the stale version and the delta is negligible, and we assume that the two versions are the same size, then we should continue to send the stale version any time the remaining stale data plus the size of the delta is less than the size of the current version. (In practice, we will know the size of the current version if the `Content-length` header field is present.) Assume the length is  $L$  and the size of the delta is  $\delta L$ . If we have already transferred  $\delta L$  bytes of the stale version, then transferring the remainder plus  $\delta L$  will require no more bytes than sending all  $L$  bytes of the current version.

Since we have no way of knowing how large a particular delta will be, any scheme that depends on computing the delta only after the whole response has been received by the server proxy can sometimes perform badly. However, there is an entire family of increasingly sophisticated abort schemes that one can think of, which can be integrated into the process of receiving the current version, producing the delta on the fly, and aborting if the delta appears large.

## 5 Experiments

Ideally, we would like to perform a long-running experiment that would compare end-to-end performance of our optimistic delta mechanism with existing proxy

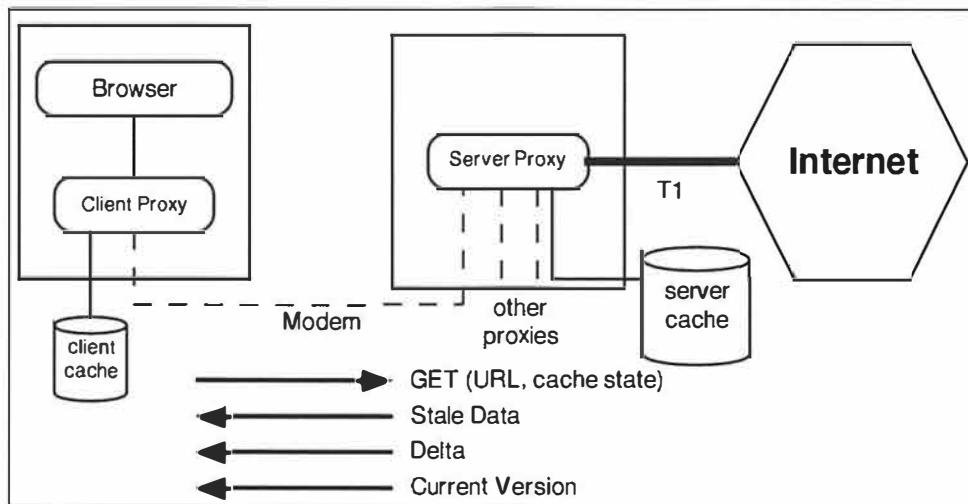


Figure 6: System architecture.

| Client proxy has cached copy? | Server proxy has cached copy? | Cached copy current? | Content provider's copy Modified? | Server proxy's action   |
|-------------------------------|-------------------------------|----------------------|-----------------------------------|---|
| <i>irrelevant</i>             | <i>no</i>                     | <i>irrelevant</i>    | <i>irrelevant</i>                 | retrieve current version and send to client proxy   |
| <i>no</i>                     | <i>yes</i>                    | <i>yes</i>           | <i>irrelevant</i>                 | send cached copy  |
|                               |                               | <i>no</i>            | <i>no</i>                         | send cached copy upon request, confirm current after GET If-Modified-Since to content provider        |
|                               |                               |                      | <i>yes</i>                        | send cached copy upon request, send delta or new copy after GET If-Modified-Since to content provider |
| <i>yes</i>                    |                               | <i>yes</i>           | <i>irrelevant</i>                 | confirm current   |
|                               |                               | <i>no</i>            | <i>no</i>                         | confirm current after GET If-Modified-Since to content provider                                       |
|                               |                               |                      | <i>yes</i>                        | send delta or new copy after GET If-Modified-Since to content provider                                |

Table 1: Possible states when requesting a URL. "Cached copy current?" refers to whether the server proxy can respond using its cached copy without consulting the content provider.

caching. One way to perform such measurements would be to get a set of random URLs, and then request each URL periodically for an extended time using the optimistic delta approach and the existing approach, and compare the average response time. We plan to perform such experiments in the future.

To date, we have focussed on "microbenchmarks" to study the two extremes of interest: the case when the client does not have a page cached, and must obtain a full copy of the page (possibly a stale copy followed by a delta or confirmation that it is current); and the case when the client and server have the same copy cached and the server can send a delta. We compare each of these against the response time of an unmodified system that connects directly to the server proxy and does not use deltas.

## 5.1 Experimental Setup

We performed our tests using a pair of Intel-based systems running the proxy code and a Sun SparcStation providing content. Specifically, the client system was a Pentium 133Mhz based machine with 32MB of RAM, running BSDI's BSD/OS v2.1. (Our proxy code is very portable across Unix platforms and currently compiles without any change on SunOS, Solaris, Linux, FreeBSD and BSD/OS. Its main system dependencies are BSD sockets and *vdelta* so we expect it should be easily portable to any system that has BSD sockets and some kind of difference library that supports binary files.) It was connected using an AT&T Paradyne Comsphere 3820Plus modem at 28.8 Kbps to a dial-in server on the AT&T corporate network. The server proxy ran on an identical machine in the AT&T network one hop away from the dial-in server. The server connected to an HTTP daemon (*htd*, an internally developed server) on a uniprocessor SparcStation 20 on the same Ethernet segment as the server proxy. This provided relatively fine-grained control over the latency between the server proxy and the content provider.

The "browser" was a simple C program that fetched a series of URLs specified in a control file by communicating with the client proxy, which also ran on the client machine. The "browser" did not do any caching.

## 5.2 Test Data

Here we report a performance evaluation using a synthetic workload based on the multi-version archive of  $W^3$  pages collected by the AT&T Internet Difference Engine (described above in Section 3.1). This archive reflected the actual evolution of the pages, although it did not contain copies of every version of every page:

some pages were archived automatically once per day when changes were detected, while the majority were archived upon the explicit instruction of a user of the system.

Slightly over half of the pages had only one version archived; these reflected pages that were registered with the system but had either never changed or (more likely) were not archived automatically and had not been selected for subsequent archival by a user. We excluded these pages from the benchmark because no deltas were available. On the other hand, about 10% of the 380 pages had 10 or more versions archived, and several had 50 or more versions (the latter were all pages that were archived automatically).

## 5.3 Benchmark

The purpose of the benchmark was to examine the effect of several parameters on end-to-end latency in the optimistic delta system: delta size, server latency, and cache contents. We considered delta size by retrieving many pages with different characteristics. We examined the effect of server latency by varying the response time prior to sending data to the server proxy (see below). Finally, we evaluated the difference between sending deltas to a client with the past version of the page cached and one without it cached. (In the case of the unmodified proxy, without deltas, caching was irrelevant because each version of the page was retrieved exactly once.) All requests were made with *Pragma: no-cache*, and the pages always differed upon each retrieval. Other cases, such as when the page has not changed or the server proxy can return its cached copy, are relatively uninteresting: they either favor the optimistic approach or are equivalent between the two systems.

In each run of the benchmark, the client system retrieved each URL repeatedly, once for each version that existed. A CGI script mapped the URL into a local filename that is dependent on the next version for that URL: the first GET on */deltatest/pageN* returned */deltatest/pageN/1*, the second returned */2*, and so on. Thus to the "browser" (actually the benchmark program) and the proxies, the same URL mapped to new versions of the page upon each request.

In addition, the CGI script read a file to determine how much of a delay it should insert before responding, which was used to simulate delay in the Internet and/or on the content provider. Longer delays permit more of an opportunity for optimistic transfers of large documents but also place a larger lower bound on end-to-end latency: if a server takes a minute to respond, then it will be at least a minute before the client proxy knows it has the current version of the page. We

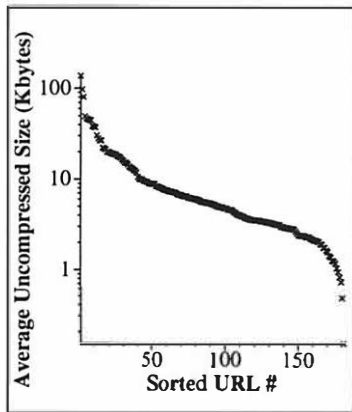


Figure 7: Average uncompressed data size across all versions of each page, sorted by size, shown on a log scale.

therefore expected that the runs with no added latency would show a high benefit from cached deltas and suffer delays from ill-placed optimistic transfers of the old copy that could not fit into the  $T_{wait}$  idle period, while runs with significant added latency would favor the optimistic approach but gain less from sending cached deltas (the amount of time saved as a fraction of total latency would decrease due to the fixed overhead).

For the current set of experiments, we were forced to restrict the range of parameters to keep the experiments tractable in a limited time frame. We did this in two respects:

- We ran experiments using fixed delays of 0s and 5s, to show extreme cases: what happens when data is nearly immediately available, and what happens when old data can be transferred during idle times.
- We restricted the maximum number of versions for a given page to 10, under the assumption that the behavior for the first 10 versions of a page would be representative of the entire set.

## 5.4 Results

To make it easier to interpret the results, we sorted URLs by the average uncompressed file size. Figure 7 graphs the average size (across all versions of a page) as a function of the sorted URL numbers, which are used in the other graphs below.

Figure 8 shows our results. Each graph plots the average ratio of end-to-end latency using the modified

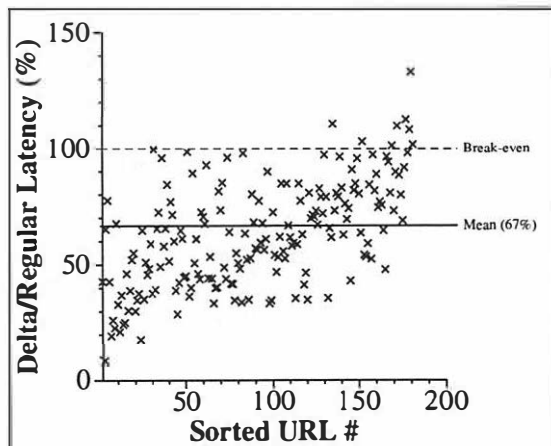
system to latency using the unmodified system<sup>4</sup>. The left column shows cases where the client proxy caches the previous version (simple deltas), while the right column shows the use of optimistic deltas. The first row shows no added content provider latency, and the second row shows 5s of added latency. The URLs are sorted in the same sequence as in Figure 7. The solid line in each graph indicates the mean of all the points in the graph, while the dashed line indicates the break-even point.

The cost of computing deltas and patching was negligible (1-2%) compared to the network transfer time and protocol processing overhead in all our experiments. Moreover, the largest measured value of overhead from computing a delta and applying the delta on the client was much less than the typical variation in the total URL fetch times.

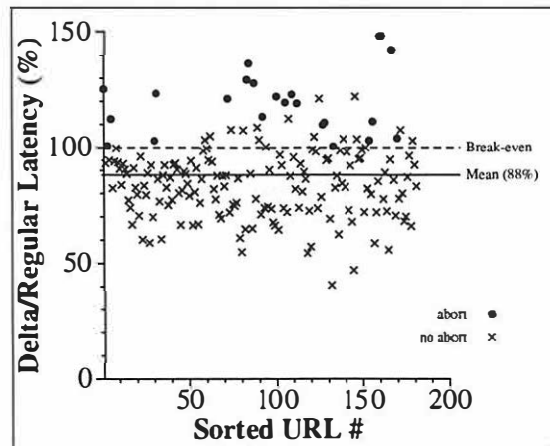
From Figure 8 we draw the following conclusions:

- The pages with the lowest index, which have the largest original file size, tended to show more improvement than the smaller pages, but although the general trend is upward as one moves right along the X-axis, there are great variations from page to page.
- As expected, without added latency, many of the pages took longer using the optimistic approach than without it. The measurements in Figure 8(b) were taken with a simple abort strategy in place. This strategy aborted only when the server proxy had finished computing the delta and the amount of remaining stale data plus the delta was more than the size of the regular response. We expect that a smarter abort strategy, such as aborting an ongoing optimistic transfer of stale data and “cutting through” new data even as it is being received if it appears that it is very different from the cached stale data, would cap the latency of our system at close to 100% of the unmodified system.
- With 5s added latency, most pages were received faster by the client using optimistic deltas, with a mean improvement of 27%. In fact, the latency for optimistic deltas with 5s added delay was consistently somewhat less than that for simple deltas with the same delay. We attribute the better performance of optimistic deltas to TCP’s slow-start algorithm [14]. In the case of the optimistic deltas the transfer of the stale data opened up the TCP congestion window, so the deltas were transferred faster in this case than in the case of simple deltas,

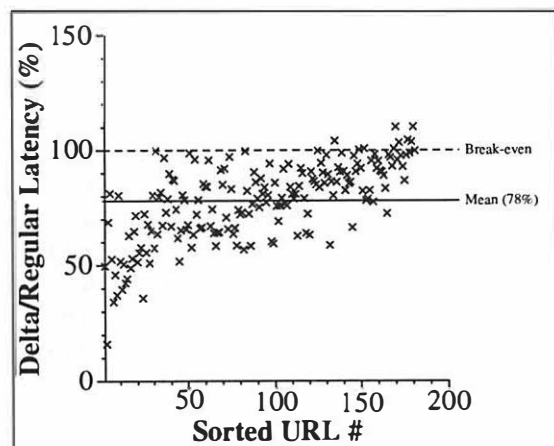
<sup>4</sup> In the unmodified system, there were no proxies involved and the client talked directly to the content-provider.



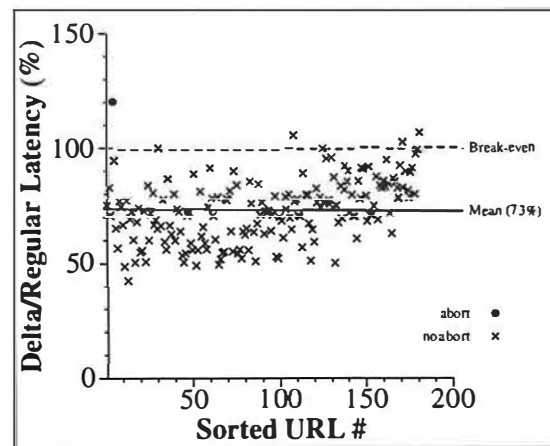
(a) Simple deltas, 0s added latency.



(b) Optimistic deltas, 0s added latency. The circles represent the 23 pages where aborts occurred.



(c) Simple deltas, 5s added latency.



(d) Optimistic deltas, 5s added latency. The circle represents the one page where aborts occurred.

Figure 8: Experimental results, showing ratios of end-to-end latency for modified versus unmodified system, varying whether old versions are cached on the client or sent optimistically by the server, and whether the content provider adds 0s or 5s of latency before returning content. Each data point represents the average across all versions for the corresponding page. The solid line in each graph indicates the mean of all the points in the graph, while the dashed line indicates the break-even point. The “simple delta” case never experienced aborts, while the “optimistic delta” case experienced aborts that are indicated with a different symbol.

where the transfer of the delta had to open up the congestion window itself.

- Nearly all of the simple deltas improved performance regardless of added latency, which one would expect. As predicted, the relative gain was generally better when the fixed overhead was lower. In the case of 0s added latency, most of the points that showed degradation were cases of only two versions being available (hence a greater likelihood of variability due to external factors) and where the deltas were 40–60% of the original file size. The overall improvement was 33%, which was the best of the four configurations.

## 6 Status and Future Work

At present, all of the functionality described in this paper has been implemented except for the ability of the server-side proxy to store multiple versions of the same URL. We plan to implement these features and test the system in a multi-user environment, where the same server proxy handles requests for multiple users. We believe that in this environment there will be many more cases where the server proxy has content that a particular client proxy does not, resulting in more optimistic transfers than would occur in a single-user context. One must evaluate policies for determining how many versions to keep and how many concurrent clients can be supported by a server proxy.

We plan to expand the URL comparison logic to handle the case of variants on the same URL (including the part of a GET URL that specifies CGI parameters), as described above in Section 3.1. In fact, it might be possible to hash the contents of pages to find other pages that are substantially similar and would generate small deltas.

Currently, each communication between the browser and the client proxy, or between the client and server proxies, requires a TCP setup. Persistent HTTP [17] should improve performance further, but we have not yet implemented a persistent connection in our proxy.

In the current system, deltas are generated only when the current version of a document has been received in its entirety. We intend to add *incremental* delta generation so that the delta can be sent over the slower link as content is received, and so that it is possible to abort optimistic transfers early if the delta appears to be large. It is also possible to use historical data to estimate the usefulness of sending stale data: if  $T_{wait}$  for a particular host or page is usually very small, then one might not bother with the optimistic transfer.

Finally, it should be useful to integrate prefetching into the optimistic delta system. In addition to prefetching new pages through the server proxy to the client (similar to the studies mentioned above [18, 22]), we can prefetch deltas to keep the proxies' caches better synchronized.

## 7 Conclusion

We have proposed an *optimistic deltas* approach to reduce the latency of accessing  $W^3$  pages. This approach involves sending the differences between versions of a page, or deltas, to the client, instead of sending entire pages. It also permits stale data to be sent during periods of inactivity. Our approach is optimistic because it sends data that may not be needed; instead, it optimizes for the common case when pages change incrementally, at the expense of a slight overhead in the rare cases when a modification drastically changes the content of the page. In other words, we assume that in most cases when a copy cached by the proxy is deemed unusable, it is either still current, or, if it has been modified, the size of the modification is considerably smaller than the page itself.

Our study of an AT&T multi-version archive of  $W^3$  pages confirmed the above assumption. In fact, by examining the extent to which the results of AltaVista queries with slightly different parameters differ, we showed that this assumption may even hold for dynamically generated pages. However, in general we expect that other sorts of data, such as images, should be handled specially rather than processed as deltas.

A study of the latency to obtain  $W^3$  pages confirmed that the latency in obtaining data may often be sufficient to send stale data, for the purpose of sending a small delta once the data is available. However, performance may be degraded when latency is low and more sophisticated techniques for deciding when to abort the transfer of stale data are required.

We implemented our approach without changing the browser. Instead, we configure the browser to connect to a *client proxy* on the same machine, which in turn connects to a server proxy. These proxies have been modified to follow the optimistic deltas approach. We compared the performance of this configuration with the original system. This performance study, based on microbenchmarks, showed a significant latency reduction achieved by our approach: an average of 12–33% improvement across all pages in the study, depending on system parameters, with some transfers improved by an order of magnitude. One particularly surprising result was the effect that transferring potentially stale data had on the TCP slow-start algorithm when a link

is otherwise idle, consistently improving end-to-end latency.

While a long-term experiment that would compare the performance of our approach with existing proxy caching systems on real-life workloads is needed, the experiments described in the paper strongly suggest that the optimistic delta mechanism results in a considerable reduction of  $W^3$  latency.

## Acknowledgments

Ankur Jain assisted with the analysis of HTTP latencies. Herman Rao wrote the initial version of the bimodal proxy used in these experiments, Dave Korn and Kiem-Phong Vo wrote *vdelta*, and Dave Kristol wrote *htd*. H. V. Jagadish initially suggested the use of stale pages in conjunction with deltas. Robin Chen, Tony DeSimone, Dave Korn, Bala Krishnamurthy, Ankur Jain, Jeff Mogul, Doug Monroe, Sandeep Sibal, and the anonymous USENIX referees provided comments on earlier drafts of this paper.

## References

- [1] Altavista. <http://www.altavista.digital.com>. Random URL selection at <http://www.altavista.digital.com/cgi-bin/query?pg=s&target=0>.
- [2] Dave Belanger, David Korn, and Herman Rao. Infrastructure for wide-area software development. In *Proceedings of Sixth International Workshop on Software Configuration Management*, March 1996.
- [3] M. Burrows, C. Jerian, B. Lampson, and T. Mann. On-line data compression in a log-structured file system. In *The Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–9. ACM, October 1992.
- [4] World Wide Web Consortium. Hypertext transfer protocol. <http://www.w3.org/pub/WWW/Protocols/>.
- [5] Adam Dingle and Tomas Partl. Web cache coherence. In *Proceedings of the Fifth International WWW Conference*, May 1996. Available as [http://www5conf.inria.fr/fich\\_html/papers/P2/Overview.html](http://www5conf.inria.fr/fich_html/papers/P2/Overview.html).
- [6] Fred Douglass. On the role of compression in distributed systems. In *Proceedings of the Fifth ACM SIGOPS European Workshop*, Mont St.-Michel, France, September 1992. ACM.
- [7] Fred Douglass. The compression cache: Using on-line compression to extend physical memory. In *Proceedings of 1993 Winter USENIX Conference*, pages 519–529, San Diego, CA, January 1993.
- [8] Fred Douglass and Thomas Ball. Tracking and viewing changes on the web. In *Proceedings of 1996 USENIX Technical Conference*, pages 165–176, San Diego, CA, January 1996.
- [9] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. In *Proceedings of the 11th Symposium on Reliable and Distributed Systems*, pages 39–47, October 1992.
- [10] Glenn S. Fowler, David G. Korn, Steven C. North, Herman Rao, and K. Phong Vo. Libraries and file system architecture. In B. Krishnamurthy, editor, *Practical Reusable UNIX Software*, chapter 2. John Wiley & Sons, New York, January 1995.
- [11] Armando Fox and Eric A. Brewer. Reducing www latency and bandwidth requirements by real-time distillation. In *Proceedings of the Fifth International WWW Conference*, May 1996.
- [12] James Gwertzman and Margo Seltzer. World-wide web cache consistency. In *Proceedings of 1996 USENIX Technical Conference*, pages 141–151, San Diego, CA, January 1996. Also available as <http://www.eecs.harvard.edu/~vino/web/usenix.196/>.
- [13] Barron C. Housel and David B. Lindquist. WebExpress: A system for optimizing web browsing in a wireless environment. In *Proceedings of the Second Annual International Conference on Mobile Computing and Networking*, pages 108–116, Rye, New York, November 1996. ACM.
- [14] Van Jacobson. Congestion avoidance and control. In *Proceedings of the ACM SIGCOMM Conference*, Stanford, CA, August 1988.
- [15] Christopher A. Kantarjiev, Alan Demers, Ron Frederick, Robert T. Krivacic, and Mark Weiser. Experiences with X in a wireless environment. In *Proceedings USENIX Symposium on Mobile & Location-Independent Computing*, pages 117–128. USENIX, August 1993.
- [16] P. Keleher, S. Dwarkadas, A.L. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of 1994 Winter USENIX Conference*, pages 115–131, San Francisco, CA, January 1994.
- [17] Venkata N. Padmanabhan and Jeffrey C. Mogul. Improving http latency. *Computer Networks and ISDN Systems*, 28(1–2):25–35, December 1995.
- [18] Venkata N. Padmanabhan and Jeffrey C. Mogul. Using predictive prefetching to improve world wide web latency. *Computer Communication Review*, 26(3):22–36, 1996.
- [19] James S. Plank, Jian Xu, and Rob Netzer. Compressed differences: An algorithm for fast incremental checkpointing. Technical Report CS-95-302, University of Tennessee, August 1995.
- [20] Bill N. Schilit, Fred Douglass, David M. Kristol, Paul Krzyzanowski, James Sienicki, and John A. Trotter. Teleweb: Loosely connected access to the world wide web. In *Proceedings of the Fifth International World Wide Web Conference*, Paris, France, May 1996.

- [21] Charles L. Viles and James C. French. Availability and latency of world wide web information servers. *Computing Systems*, 8(1):61–91, Winter 1995.
- [22] Stuart Wachsberg, Thomas Kunz, and Johnny Wong. Fast world-wide web browsing over low-bandwidth links. Available as <http://ccnga.uwaterloo.ca/~sbwachs/paper.html>, June 1996.



# A Toolkit Approach to Partially Connected Operation

Dan Duchamp

*Columbia University  
and  
AT&T Labs*

## Abstract

Partially connected operation is the circumstance in which the communication link between two computers is intermittent, either by choice or because of failure. This paper describes the design and performance of a user level toolkit that is suited to accessing the home directory of a partially connected user in a bandwidth-efficient manner.

Compared to custom file systems for partially connected operation, the toolkit is easier to deploy and provides extra degrees of flexibility because it runs at user level and uses the local file system for its cache. The toolkit makes it possible for unaltered clients to access NFS file systems exported by unaltered servers.

The maintenance of consistency between client and server is automatic provided that certain assumptions are upheld, the primary one being that sharing is limited in such a way that there is a “single locus of update.” That is, for extended periods, updates are applied either to the client's cache or directly to the server, but not to both simultaneously. This pattern of use is typical of a user's home directory.

Performance is the disadvantage of user level operation. The client's cache is managed by a “caching tool” that services every file system operation, and this redirection increases latency substantially.

## 1 Introduction

This paper explains a set of simple user level tools that is suited to accessing one's home directory over an intermittent connection. The toolkit implements “partially connected operation” for stan-

dard file system protocols (e.g., NFS version 2) running on unmodified UNIX hosts. The work was motivated by the following personal experience, which provides an example of partially connected operation.

My home computer has a leased line connection to an Internet service provider (ISP); my company uses a different ISP. In order to avoid being a system administrator, I initially used my home computer as an X terminal, `telnet`-ing to the machines at my office. Unfortunately, network response is highly variable—waiting seconds or minutes for keystroke echo is not unusual, outages are frequent (e.g., several times per evening) and sometimes long (e.g., an entire holiday weekend)—enough so that interactive operation over the Internet proved impractical.<sup>1</sup>

I considered two alternatives. One was to continue using the home computer as a terminal, dialing up to the office. The other was to make the home computer as self-contained as possible. My home is located in a different area code from my office, so dialup accrues long distance charges both per-call and per-minute; therefore, I chose the second alternative.

I implemented a “toolkit” of user-level programs that together form what might be viewed as a poor man's version of a distributed file system. The system minimizes the connect time between the client at home and the server at work, hence “partially connected operation.” Compared to the pioneering work on file systems supporting disconnected and/or partially connected operation

---

<sup>1</sup>The primary cause of the problem is that home and office have different ISPs. The two ISPs connect at the “MAE West” router on the west coast, so packets between New York City and northern New Jersey—four miles as the crow flies—travel across country and back again, typically going through 20 hops.

(e.g., [9, 14, 6, 8]), what I have implemented is unambitious and not technically groundbreaking. However, over time it became clear that operating at user level—where tools such as file system utilities, scripting languages, etc. are available—brings some profound advantages. My toolkit approach stands in contrast to the approach based on custom kernels and file system protocols. My claim is that a simple user-level toolkit can handle the most common workload (i.e., files drawn from a personal home directory, with updates coming from only one side of a connection at any time) with considerably less mechanism, more flexibility, and much greater ease of deployment.

In my approach, the home computer is a file system client that is disconnected from the servers at work except when a phone call is made between them. The derived technical problem is how to keep the home and office copies consistent while minimizing phone charges. The toolkit is used as follows.

1. The office file server (master) exports my home directory to the home computer (slave) as a separate NFS file system.<sup>2</sup>
2. The slave caches a subset of my home directory and also has a normal collection of slowly changing system files. Cached files are kept not in a hidden area, but in the local file system, in the same hierarchy, with the same names, as at the master. When I am at home applications execute on the slave, accessing files using the same names as if the application were running at the office, but the files come from the cache.
3. The slave fetches files and directories on demand, using the NFS protocol (version 2); however, it does not mount the master's exported file system in the usual way. Instead, the master file system is mounted so that a local process is its NFS server. Whenever the slave accesses a file with a name that is in my home directory, the process—called the “caching tool”—receives control. When possible, it redirects the access to the local cached file with the same name. If the relevant file is not yet cached, the process fetches it and copies it to the local cache.

The caching tool is a modified version of Amd [17] (version “upl 102”), a user level auto-

<sup>2</sup>At my site, this required some administrative changes. Ordinarily, several home directories are bundled into a single file system.

mounter. Partially connected operation is implemented as a new file system type (named “cfs”). A UNIX client need only upgrade to the new version of Amd to get partially connected access to home directories as well as all other services currently supported by Amd.

4. When I move between office and home, the file hierarchy at the new location will be out of date with respect to the old location if I made changes on the old system. To determine the changes to propagate from the old system to the new, I run on each system a “checksum tool” that computes cryptographic checksums for the file hierarchy on that system.

The checksum of a directory is computed over the names and checksums of the directory entries. If two corresponding directories have the same checksum, the entire hierarchies rooted at those directories are identical.

5. The checksum tool produces a report that is fed into a “comparison tool.” This tool uses the report plus knowledge of which site has the most recent changes to decide which files and directories to copy to or delete from the site that is out of date.

The comparison tool achieves massive pruning from the fact that equality of two directory checksums implies equality of the entire hierarchies rooted at those directories.

This approach does not constitute a traditional distributed file system. Rather, it is a toolkit appropriate only in the case where there is an easily identified “single locus of update”—i.e., for extended periods, updates are applied either to the home file system or to the office server, but not to both. The checksum and comparison tools are invoked whenever the locus of updates changes, and together they reestablish consistency. Neither file system should accept updates while the checksum and comparison tools are running. Since home systems and portable computers are intimately personal, the limitation might not be as severe as it may seem at first. Indeed, experience with the Ficus file system in a similar setting of a home workstation suggests that there is only a single locus of updates—that “the user acts as a write token” [4, 18].

While the single locus of update restriction significantly reduces the technical problem addressed, the toolkit nevertheless is interesting for a few practical reasons. First, the one case it handles (a single easily identified locus of updates)

is a common, and perhaps the dominant, read-write file system workload. Further, no kernel changes are required at either client or server, and only a few configuration changes are needed. The major ones are, first, to have the server export the home directory as a separate file system, and, second, to make the caching tool service the mount point at the client. All the tools run at user level; only the caching tool must run with root privilege. A consequence of the easy deployment is that for the first time standard file systems (e.g., NFS) are available to ordinary, partially connected UNIX clients. Finally, this work is highly portable. The checksum and comparison tools are “ordinary UNIX programs” that need no special privilege. The most complicated component, the caching tool, is a version of Amd, which is one of the most widely ported UNIX programs in existence. The caching tool inherits this high degree of portability. Perhaps ironically, the tool that is by far the hardest to port is the dialup software.

The next section explains the operation of the tools. The section after that presents performance analysis. Section 4 discusses related work.

## 2 Tools

The next few sections explain the design and operation of the tools for checksum computation, file hierarchy comparison, caching, and dialup.

### 2.1 Checksum Tool

The checksum tool writes a file — named “.contents” — in each directory of a hierarchy. Such a file contains the type of checksum algorithm used and the names and checksums of each file, directory, or symlink in the directory as of the time the tool was invoked. The name/checksum entries are sorted by name. Other objects that can populate directories (like devices and sockets) are ignored. Any pre-existing .contents file is also ignored.

The checksum of a symbolic link is computed by *not* following the link but rather by checksumming the contents of the link. Not following symlinks converts a file system hierarchy from a general graph into a DAG. The DAG is then treated as a tree; i.e., a file with multiple links appears several times in the tree, once in each directory. The checksum of a directory is the checksum of its .contents file; therefore, checksumming is a bottom-up operation.

As an example, suppose a directory contains two files *w*, *x*, a symlink *y*, and a directory *z*, with checksum values 1, 2, 3, and 4, respectively. The following text would be checksummed and the resulting value considered to be the checksum of the directory:

```
algo MD5
w 1
x 2
y@ 3
z/ 4
```

In this case, 3 is the checksum value of the symlink stored in *y*. The “@” and “/” symbols denote symlinks and directories, respectively. The comparison tool uses this type information, as it may need to treat symlinks and directories differently.

The available checksum algorithms are MD4 [19], MD5 [20], and UNIX “sum.” The checksum algorithm is a modular component of the tool, and it is trivial to add new choices.

An entry that is unreadable is considered not to exist and is not listed in .contents. If a directory is unwritable, then its checksum field is left blank in the .contents file in the directory above.

If the two file hierarchies to be compared are on hosts linked by a distributed file system (such as NFS), then the comparison tool can simply read the .contents files in corresponding directories of the two hierarchies and perform its comparisons based on what it reads. However, in some environments it may be impossible or impractical for the two hosts to be linked by a distributed file system protocol; instead, some other protocol (such as FTP) is used for short-lived and/or limited data shipping. To support such environments, the checksum tool optionally can also dump all directory checksums — and *only directory* checksums — into a “dumpfile.” This file, which is expected to be relatively small compared to the entire hierarchy it summarizes, can then be sent from one host to the other, and the receiving host invokes the comparison tool, which operates as described below.

### 2.2 Comparison Tool

The comparison tool has a few different modes of operation. The command line specifies the mode as well as which hierarchy has been the most recent locus of updates.

The checksum tool may have written .contents files or a dumpfile, and it may or may not be possible for the comparison tool to access both hier-

archies via normal UNIX calls such as `open()` and `fread()` (i.e., both hierarchies are accessible as file systems).

Regardless of which case is exercised, comparison of checksums proceeds top-down, with every match pruning the comparison. In the most extreme case, the checksums of the root directories of the two hierarchies match, and a single comparison concludes that the entire hierarchies are identical. More typically, one or both sides will have entries that the other side does not have.<sup>3</sup> In these cases, and in cases where both sides have an entry but with different checksums, the comparison tool emits UNIX commands that make updates, creates, and deletes in order to bring the outdated hierarchy up to date.

Large files with holes are handled specially, as explained in Section 2.3.3: a special copy program is used to transfer only sections of a large file.

### 2.2.1 Precious Files

The term “precious file” is taken from `rdist` [1]. At my site there is one significant exception to the single locus of updates behavior. We use `hlfsd` [26] to deliver Email into a user’s home directory.<sup>4</sup> Regardless of whether the user is at home or work, Email goes into a file on the server at work.

Special purpose protocols like POP3 [21] or IMAP4 [2] exist to remotely manipulate mailboxes, so the comparison tool does not change the mailbox (and any other such precious files), leaving them for the special protocols.

### 2.2.2 Fault Tolerance

Assuming that the single-locus restriction is maintained and that checksumming and comparison is done between locus changes, a failure during checksumming or comparison impacts performance but not correctness.

There is no state kept outside the client and server file systems. The checksums are derived information, and the copying done by the comparison tool is idempotent assuming that non-precious files are not updated while the comparison tool runs. Therefore, if there is a failure during the

<sup>3</sup>The `.contents` files are sorted in order to speed the task of detecting entries that are in one `.contents` file but not the other.

<sup>4</sup>`hlfsd` controls the name space of the system mail spool area — e.g., `/usr/spool/mail` — and converts the name of each user’s mailbox into a symlink into the user’s home directory. For example, `/usr/spool/mail/user` becomes a symlink to `$HOME/.mailspool/user`.

checksum/comparison phase, restarting from the beginning will establish a correct state.

The two real dangers are violating the single-locus-of-update restriction and making updates during the checksum/comparison phase. Section 2.4 discusses the steps that can be taken against these dangers.

## 2.3 Caching Tool

The caching tool is an altered version of the latest release of the Amd automounter, a program written by J-S. Pendry [17].<sup>5</sup> Many of the alterations were taken from the Autocacher, which was written by R. Minnich [12]. The Autocacher is itself an altered version of an old release of Amd. I ported the Autocacher features into the latest version of Amd, then added my features.

This section explains the new features, assuming that the reader understands how Amd and Autocacher operate. Readers lacking this background can find it in Appendixes A and B.

For file systems of type “cfs” (a name taken from Autocacher, denoting “caching file system”), the caching tool implements all NFS operations. The primary data structures and the implementations of `read`, `readdir`, `readlink`, `lookup`, and `getattr` are largely those ported from the Autocacher. The new operations are “mutating” operations that make updates to the local cache; they can be classified into categories: delete (`remove`, `rmdir`, `rename`), create (`mkdir`, `create`, `rename`), and update (`setattr` and `write`). Note that the `rename` operation falls in two categories.

It is undesirable to have every file system operation leave the kernel to go through the caching tool; Section 3 shows the negative impact on performance. However, redirection of all operations through the caching tool is necessary, for these reasons:

1. The caching tool needs to know about deletes.
2. Because of the way deletes are represented in the file system, the caching tool must also know about creates. If a create operation recreates a deleted name, the delete indicator must be deleted. Section 2.3.2 elaborates.
3. The caching tool must be in control of read operations in order to intelligently transfer large files (see Section 2.3.3).

<sup>5</sup>J-S. Pendry wrote the original Amd. Erez Zadok merged patches submitted by many persons and into the latest version, named “upl 102.” [25]

These reasons justify the caching tool handling some, but not all, NFS operations. It handles all operations because there is no way — without special kernel support such as stackable vnodes [5] — to selectively redirect a subset of operations through the caching tool while letting others go directly to the native file system. The caching tool must return a file handle on every lookup, and this handle is used for all operations thereafter.

There are four interesting aspects to the implementation: copy policy, deletes, large files, and cache management.

### 2.3.1 Copy Policy

The rule, taken from Autocacher, is that files are copied from the master lazily but directories are copied eagerly. This means that a lookup on an uncached directory results in the directory being copied and the returned handle pointing to the local copy, whereas a lookup on an uncached file does not result in a copy. The returned file handle is one fabricated by the caching tool and used for emulation. Files are cached lazily because it is common for a file to be looked up but not accessed. A lazy caching policy avoids transferring every file that is shown by an `ls` command, for instance.

The decision to treat files and directories differently is founded on two assumptions. First, that the number of directories and the number of bytes occupied by directories are both small compared to the same numbers for files. Second, that the directory structure of a user's home area changes slowly. The first assumption implies that it does not cost much to copy uncached directories to the client. The second assumption implies that, once cached, a directory is likely to stay at the client.

### 2.3.2 Tracking Deletes

The delete operations require some care. The reason is that files and directories are demand-fetched from the master, meaning that the slave has only a subset of the master's contents. So when the comparison tool finds that a file or directory exists at the master but not at the slave, there is potential ambiguity: has it been copied to the slave and deleted there, or never copied to the slave?

I considered two approaches to resolving the ambiguity. The first is to perform every delete at the master as well in the slave's cache. This removes the ambiguity: a file or directory present at the master and absent at the slave has not been copied to the slave. However, this approach increases the number of interactions between slave and master,

which is potentially costly if the network connecting the two is pay-per-use.

The second approach, which I used, is to maintain a delete log. With this approach, the ambiguity is removed in the opposite way: if a file or directory is present at the master and absent at the slave, it has been deleted at the slave if and only if the slave has a record of the deletion.

I maintain what might be considered a distributed log: deleted files and directories are renamed in the cache. The renamed files and directories are called "deletion markers."<sup>6</sup> As shown in Figure 1, deleted files are truncated and renamed with a special prefix. A deleted directory is similarly renamed provided that it contains only deletion markers. Before being renamed, the deletion markers in the directory are all deleted; Figure 2 illustrates.

The create operations make new directories and files as needed in the local disk cache, but they must be aware of deletion markers. When create creates a file, it checks for the associated deletion marker and deletes it if found, then creates the file. Checksumming will detect any conflict between the new file in the cache and the old file back at the master. Other interactions between create and delete operations are more subtle.

One subtle interaction is that deletion markers must be removed from a deleted directory so that later operations will proceed correctly. Suppose otherwise: that a deleted directory were simply renamed as a marker, and the markers within it were preserved. If the same name is later `mkdir`-ed and `rmdir`-ed again then the `.deleted.DIR.` name will exist and will not be empty at the time of the second `rmdir`. In this case the rename will fail because the marker directory will not be empty.

A second subtle interaction between creation and deletion is that in contrast to `create`, `mkdir` *does not* delete an existing deletion marker that corresponds to the name it is creating. Once a directory has been deleted at the slave, its deletion marker remains, even if the directory is re-created. The deletion marker will remain until it is deleted by the comparison tool. If a directory's deletion marker were deleted when it is re-created, then the sequence `rmdir foo ; mkdir foo` would result in a state indistinguishable from not having cached the files in `foo`, and the comparison tool would fail to delete the files on the master. Retaining both the new directory and the deletion marker indicates that it was deleted and recreated. Delet-

<sup>6</sup>A similar mechanism is the "whiteout" used for 4.4BSD union mounts [16].

---

Before deletion:

```
-rw-r--r-- 1 djd faculty 7183 Jun 14 22:26 foo
```

After deletion, foo is gone and a marker remains:

```
-rw-r--r-- 1 djd faculty 0 Jun 14 22:26 .DELETED.foo
```

Figure 1: File Deletion Using Markers

---

---

If before deletion:

```
drwxr-xr-x 2 djd faculty 512 Aug 17 1995 src
```

contains:

```
-rw-r--r-- 1 djd faculty 7183 Jun 14 22:26 foo
```

```
-rw-r--r-- 1 djd faculty 0 Jun 14 22:26 .DELETED.bar
```

Then rmdir will return an error because foo still exists.

However, if before deletion the contents are:

```
-rw-r--r-- 1 djd faculty 0 Jun 14 22:26 .DELETED.foo
```

```
-rw-r--r-- 1 djd faculty 0 Jun 14 22:26 .DELETED.bar
```

Then rmdir will succeed and result in the empty directory:

```
drwxr-xr-x 2 djd faculty 512 Jun 14 22:27 .DELETED.DIR.src
```

Figure 2: Directory Deletion Using Markers

---

ing the re-created directory requires removing the directory while leaving the deletion marker.

### 2.3.3 Large Files

As described to this point, emulation is accomplished by copying the whole file from master to slave, then having the slave's caching tool operate on the local copy. If the file is large, and the communication link is slow and/or costly, copying the whole file can be expensive and cause substantial delay. For example, copying a 1MB file over a telephone line with effective throughput of 2.7KB/sec—75% of 28.8Kb/sec, an optimistic scenario—takes well over 6 minutes. Accordingly, the toolkit handles large files specially, transferring, checksumming, and comparing smaller units.

When the caching tool “copies” a large file in fact what it does is copy the file into a special marker file in the same directory and make the file's name be a symlink pointing to the marker.<sup>7</sup> The caching tool notes that a file is a large one and handles reads and writes specially. Read operations transfer only the requested byte range, as-

<sup>7</sup>The name of the marker file is the file's name preceded by “.large-file.”.

suming it is not already cached. Write operations also copy the requested byte range, if necessary, then apply the update.

The physical file in the cache is kept in a special format. The caching tool writes a header region at the beginning of the file that describes what ranges of the master file have been copied into the cache, and at what offset each range can be found in the cached file. (This special handling of large files is an example of the flexibility that can be achieved with user-level tools.)

When the checksum tool runs, it will always compute different checksums for the large file's name at client and server, because the name will be a file at the server but a symlink at the client. When the comparison tools detects a checksum mismatch, and detects that the client name is a symlink whereas the server name is a file, it then inspects the symlink value to determine if it is a marker for a large file. If it is, then a special copy utility reads the header region of the cache file and copies to the master file only the byte ranges that the cache file holds.

The definition of a “large” file is a parameter that can be set either on Amd's command line or via `amq`, the configuration utility for Amd. The

parameter is a number of seconds. The caching tool measures the network throughput achieved by recent copy operations. A file is “large” if copying it would take more than the specified time.<sup>8</sup> By setting the time parameter to zero, the user can treat all files as large files. The reason not to do is that the checksum/compare phase will take substantially longer than usual if every file is a checksum mismatch.

### 2.3.4 Cache Management

Storing the cache and the “log” in the local file system contrasts with the approach taken by specialized in-kernel file systems that implement the log and cache as separate data structures, both outside the file system name space.

With the cache in the local file system, ordinary utilities and scripts can be used as cache management tools. This toolset can be superior in flexibility, scope, and ease of use compared to the mechanisms that file system implementors can include into the client side of an in-kernel file system. Therefore, it may be easier to implement complex, user-specific and/or time-varying criteria for removing files or handling special cases. For example, cache files can be deleted based on any criteria any time the cache is consistent with the master. (I put cache cleanup in the logout script before the checksum phase.)

Coda and AFS both maintain a log of all mutating operations. The most important purpose for a log is to record deletes, but log information is also exploited for attempted automatic resolution of conflicts. In both systems, the log is a single data structure, and managing it is a source of complexity. If the log fills, processing stops; however, for typical workloads, information early in the log is superseded by later log entries. This motivates techniques such as log compression [7] and “trickle discharging” [14]. In my approach, the “log” and the cache share the same storage, so neither has a size limit that might not be ideally matched to the other’s size.

## 2.4 Change of Locus

Change of locus raises three issues: how to detect it, exactly when to execute checksum operations, and how to detect that the single-locus-of-updates assumption has been violated.

<sup>8</sup>The first file, accessed before there is any network throughput information, is deemed large if it exceeds 50KB.

Change of locus is detected by running a script. The script runs when the user arrives at the new site or departs the old site. It can be executed by the user or can be invoked automatically as part of login/logout or as part of a terminal locking program, etc. The tradeoff between running the script at the new or old site is performance versus safety. If the script is written assuming that the user is leaving his old site, the checksum operations can be overlapped with the user’s movement, but there must be some means to indicate that the script is finished (because the user must refrain from making updates at the new site until the compare/copy operation is finished). I have not implemented such a scheme, instead doing one checksum operation at logout and the other at login.

The compare phase takes time proportional to the number of updates at the most recent locus. The checksum phase takes time proportional to the size of the file hierarchy, and the constant factor is much larger than for comparison. Therefore, checksumming the master is by far the longest operation. Accordingly, I use login/logout as the indication of locus change, checksumming the master at logout and running the slave checksum and comparison at login.

Invocation of checksumming at the other site is accomplished with `rsh`.

Violation of the single-locus assumption can be detected simply. In particular, when checksums are computed at the new locus, existing `.contents` files should not be overwritten. Instead, they should be moved, say, to `“previous-contents.”` Then the comparison tool not only compares corresponding `.contents` files at the two locations, but also compares `.previous-contents` and `.contents` at the new locus. If they disagree, the single-locus assumption was violated; i.e., updates were made at both the new and old loci.

Note that consistency re-establishment need not occur only at change of locus. That is, a home user can invoke the tools in the middle of his session *as if* a change of locus were underway, provided that he refrains from updates for the necessary period of time.

## 2.5 Communication Software

Although dialup is a mundane function, the dialup package is the greatest obstacle to the toolkit’s portability and deployment.

Dialup is a portability problem because what

is needed is *on demand* dialup. That is, a telephone connection must be brought up automatically whenever there is outgoing traffic; dialing upon login, say, and holding the connection defeats the purpose of the toolkit. Many UNIX TCP/IP stacks will simply return an error if there is not already an active link device ready to accept outgoing traffic. Providing a hook to allow a user level script to set up a link on demand is an operating system issue, thereby making it hard to formulate a solution that is as portable across UNIXes like Amd. My home client runs Linux. Linux has the `diald` on-demand dialup package [24] that provides the right function. This is what I use.

Dialup can be a deployment problem because machines on both sides of the communication link need to dial each other. Most installations have firewalls, many have restrictions on acceptable sources of dial-in and targets of dial-out, and many elect not to support SLIP or PPP access. These restrictions can be overcome, but may increase the amount of administrative changes that must precede using the toolkit.

An issue for the script that sets up and tears down the telephone connection is when to tear down. In a slightly different context this has been called the "Holding Time Problem" [22]. The issue is that maintaining an open connection costs money, but so does restarting a closed connection, so shutting down immediately is not necessarily the best policy. Tariffs are known and, if future traffic patterns were known, a simple calculation would indicate how long an open connection should be maintained. However, there is no realistic way to predict future traffic, especially from a single host. Accordingly, I have changed `diald` to include the heuristic that the telephone connection is shut down after an idle period whose connect charge equals the charge to make a new phone call. This policy limits the overall cost of a phone call to twice the charge to make the call (the real charge for making the call plus the charge for the idle time that causes the shutdown) plus the charge for the time used. Here time "used" means real use plus all idle periods shorter than necessary for a shutdown.

It is desirable to replace the dialup package with a more powerful "communication tool" that could direct traffic to my Internet connection opportunistically, falling back to the phone line only when the Internet connection becomes unacceptable. Such a tool is even more operating system specific than a dialup package. Work is underway

---

| TYPE OF LOOKUP         | LATENCY  |
|------------------------|----------|
| NFS, uncached          | 6.4 msec |
| NFS, cached            | 2.7      |
| Caching tool, uncached | 49.3     |
| Caching tool, cached   | 49.2     |
| Cachefs, uncached      | 10.4     |
| Cachefs, cached        | 3.4      |

---

Table 1: Cost of Redirection, Lookup

on a prototype.

### 3 Evaluation

The next few sections report performance measurements. Section 3.5 summarizes the administrative changes that might be needed to deploy and use the toolkit on UNIX.

#### 3.1 Basic Cost of Redirection

The first measurement determined the basic cost of redirection. I did a number of `getattr` operations, measured the overall time to do them as well as the time spent in the caching tool, then subtracted. The result is the amount of time it takes to go into the kernel, have the kernel redirect the NFS operation to the caching tool, which acts as the NFS server, then return the result.

On a Sun SparcStation-2 running Solaris 2.4, the result is 38.1ms with a standard deviation of 3.2ms.

#### 3.2 Added Cost of Emulation and Caching

The second experiment determined the time needed to do certain common NFS operations such as `lookup` and `read`. The caching tool was timed as was regular NFSv2 and Sun's "cachefs" file system, an in-kernel file system that caches on disk.

In the lookup test (Table 1) the parent directory had already been looked up. The reported times are end-to-end measured from a user level program. An "uncached" lookup goes to a remote file server. In each test, the standard deviation was very small.

In the read test (Table 2) the file had already been opened and one 4KB block was read. The reported times are end-to-end measured from a user level program. An "uncached" read goes to

---

| TYPE OF READ           | LATENCY  |
|------------------------|----------|
| NFS, uncached          | 6.7 msec |
| NFS, cached            | 5.4      |
| Caching tool, uncached | 175.2    |
| Caching tool, cached   | 76.2     |
| Cachefs, uncached      | 26.7     |
| Cachefs, cached        | 21.1     |

---

Table 2: Cost of Redirection, Read

a remote file server. A cached read is drawn from memory in the case of NFS, and from disk in the case of the caching tool and Sun's Cachefs. In each test, the standard deviation was very small.

The very slow times for the caching tool show the effect of copying data between a process and the kernel. For the uncached read via the caching tool, the data is copied from the remote server to the caching tool, written from the caching tool into a local file, then copied back into the kernel as the result to return to the requester.

### 3.3 Checksum Speed

I ran the checksum test with my home directory as input. My home is on a faster platform (SparcCenter-1000 running Solaris 2.5). I measured 4.6ms per file with MD4 and 6.2ms per file with UNIX sum. For my home directory of 7502 files comprising 899MB, it takes MD4 close to 35 seconds to compute checksums.

### 3.4 Accuracy of Single-Locus-of-Update Assumption

The effectiveness of the toolkit rests, in large part, upon the assumption that home directories typically have a single locus of updates except for a few easily identified exceptions such as Email delivery.

Numerous studies have shown that there is very little write sharing in file systems; for example, [9] examines the number of times consecutive updates are made by different users. The data is suggestive, but "different users" is not the issue. It could be that the same user is making consecutive updates, but from different sites. If one of these sites becomes disconnected yet still makes updates, there would be conflict even though the same user made all the updates.

The toolkit requires the stronger condition that while updates are made at home they are not made at the office server, and vice versa, for the user's

entire home directory. I am aware of no study that measures usage patterns with sufficient precision to decide how dangerous the single-locus-of-update assumption is. One complicating factor is that it is acceptable for updates to come from any number of office-side client machines during the time when the locus of updates is at the office.

To shed some light on the matter, I changed the office-side mount daemon so that it timestamped its records of remote mount/unmount operations. Using the time information one can tell at all times how many clients had a file system mounted. I have seen no unexplained mounts during the time when home is the locus of updates. However, as discussed earlier, it is possible to detect when a file had been changed at both sites by performing one extra checksum operation and one extra compare.

### 3.5 Summary of Administrative Actions

The following system administration steps must be taken to setup and use the toolkit for partially connected access to one's home directory. Depending on an installation's current policies, these steps may or may not represent changes.

1. At the server: make the user's home directory a separate file system, and export this file system so that the user's home machine can mount it.
2. At the client: install the caching tool as the automounter; the caching tool can replace Amd version upl 102.
3. At the client: add the "cfs" entry to the Amd maps, and allocate an area of the local file system to serve as cache and log.
4. At the client: install a dialup-on-demand package.
5. At both sites: allow IP dial-in and/or dial-out to the other site. Allow execution of rsh commands sent from one site to the other.
6. The user: write scripts to invoke checksum, comparison, and cache cleanup. Execute these scripts from login/logout or other scripts that execute manually or automatically when the locus of updates switches.

## 4 Comparison to Related Work

There are two principal categories of related work. One category contains the many commercial products for "file synchronization." Well known products include Symtanec's *PC Anywhere*, Traveling Software's *Laplank*, and Microsoft's *Traveler's Briefcase*. The other category is the published work on "normal" distributed file systems that have been extended to accommodate users who suffer various levels of disconnection. In this category are Coda [9, 10, 11, 13, 14, 15, 23], Ficus [3, 4, 18], and the Michigan alterations to AFS [6, 7, 8].

My work sits in between. The main difference with the commercial tools is that operation is more automatic, no special software is required on the master server, or both. Also, I know of no tools, commercial or otherwise, that use hierarchical checksums. The main difference with the research file systems—besides scope, a special purpose toolkit versus a full fledged distributed file system—is point of view. Perhaps because of the history of the general purpose file system projects, the point developed in many of the papers cited above is that general purpose techniques can be extended and adapted to roughly accommodate the needs of (completely, partially, and mostly) disconnected users. This paper raises and partly answers an opposing question: can a large fraction of the usefulness of distributed file systems for partially disconnected users be delivered through a portable toolkit that has only a small fraction of the complexity? The papers cited above explore whether extension of existing distributed file system techniques is *sufficient* to accommodate disconnected users; this paper raises the issue of whether such techniques are *necessary*.<sup>9</sup>

The main issue is the workload. If the workload upholds the single-locus-of-update assumption, then the consistency problem becomes trivial and the need to build support for consistency protocols, logging, and conflict resolution into the basic file system becomes questionable.

It is widely accepted that most areas of a file namespace can be accurately characterized as ei-

ther *personal* read-write areas or read-only shared areas. Data gathered from the file system projects suggests that personal areas typically do have only a single locus of updates, and that conflicts arising between replicas can be handled simply, as the comparison tool does. For example, [4] states:

Another factor contributing to the rarity of conflicts is the effect of a human write token ... Because updates to personal data come primarily from a single user, that person serves effectively as a "write token" for those files. By arranging a pattern of reconciliation corresponding to the presence of the user, the most recent data is almost always present when updates are made.

The "arranging" referred to is the tweaking of system parameters that govern when the reconciliation function built into the file system runs. One might ask whether—given a fast user level reconciliation procedure—reconciliation should be done that way: by the file system, with involvement of the system administrator. Further, [15] states:

In the vast majority of cases, resolution merely involves overwriting a stale replica with the most current one.

A third category of relevant work is techniques to automatically resolve conflicts [10, 18, 11]. This work is valuable in its original context and is complementary to my work: a slight generalization of the comparison tool would have it invoke such tools whenever a conflict is found. Operating at the user level allows such tools to become quite elaborate, if necessary.

A final point regarding similar tools is that the function provided by the toolkit cannot be duplicated simply by *rdist*-ing from one site to another. The difference is that the caching tool copies on demand from the master and tracks deletes. Copying on demand is more bandwidth-efficient, and the tracking of deletes allows the comparison tool to propagate deletes from one site to another.

## 5 Summary

I have described a small set of tools that can be used, in certain circumstances, to maintain consistency between a master set of files kept on a

<sup>9</sup>There is middle ground. The "fetch only mode" in AFS [8] is a step in the direction of my work. In fetch-only mode fetches are made but the log is not replayed until a later time. Since log replay is what detects (and maybe resolves) conflicts, not doing it results in operation similar to mine of fetching on demand but not checking for conflicts until a locus change.

server and copies kept on a partially disconnected client.<sup>10</sup> Because only limited server-side system administration changes are required, users can decide individually whether to use the toolkit.

The work makes two main contributions. First, the software enables partially connected users to access their NFS home directories from stock UNIX clients. Second, the toolkit design provides a stark contrast to earlier work on custom file systems for partially connected operation. The toolkit is not as fast, not as general, and may depend on outside forces to maintain the validity of key assumptions; however, the implementation at user level makes the work more portable and provides many degrees of freedom in selecting how the toolkit operates.

A partial list of “such degrees of freedom” is: when to perform checksums, whether to test for violation of the single-locus-of-updates assumption, how large is a “large file,” user-programmable cache cleanup, ability to invoke programs that automatically resolve conflicting updates, and how to manage the dialup connection.

## 5.1 Future Work

One obvious area for future work is difference copying. A number of commercial packages exist for Windows and provide function similar to the toolkit. Most of these packages provide for copying only differences in changed files. This is a complication, as more state must be kept on both sides. My personal experience so far has been with small text files, so the compare/copy phase runs fast enough. But in different settings differencing could be crucial.

### Availability

The toolkit is available at <http://www.mcl.cs.columbia.edu/src/cfs>.

## 6 Acknowledgements

This work was performed while the author was in the Computer Science department at Columbia University. The work was supported in part by ONR grant number N00014-93-1-0315, and in part by DARPA order number B094, monitored under ONR contract number N00014-94-1-0719.

<sup>10</sup>The “disconnection model” is that connection by telephone is expensive and to be used frugally, but is always available on demand. There are other disconnection models.

## References

- [1] M.A. Cooper.  
Overhauling Rdist for the '90s.  
In *Proc. Sixth Systems Administration Conf. (LISA VI)*, USENIX, pages 175–188, October 1992.
- [2] M. Crispin.  
Internet Message Access Protocol – Version 4.  
RFC 1730, IETF Network Working Group, December 1994.
- [3] R.G. Guy et al.  
Implementation of the Ficus Replicated File System.  
In *Proc. Summer 1990 USENIX Conf.*, USENIX, pages 63–71, June 1990.
- [4] J.S. Heidemann, T.W. Page, R.G. Guy, and G.J. Popek  
Primarily Disconnected Operation: Experiences with Ficus.  
In *Proc. Second Wkshp. on the Management of Replicated Data*, IEEE, pages 2–5, November 1992.
- [5] J.S. Heidemann and G.J. Popek.  
File-System Development with Stackable Layers.  
*ACM Trans. Computer Systems*, 12(1):58–89, February 1994.
- [6] L.B. Huston and P. Honeyman.  
Disconnected Operation for AFS.  
In *Proc. USENIX Mobile and Location-Independent Computing Symp.*, USENIX, pages 1–10, August 1993.
- [7] L.B. Huston and P. Honeyman.  
Peephole Log Optimization.  
*IEEE Computer Soc. Bull. Tech. Cmte. on Operating Systems and Application Environments*, 7(1):25–32, Spring 1995.
- [8] L.B. Huston and P. Honeyman.  
Partially Connected Operation.  
In *Proc. Second USENIX Symp. on Mobile and Location-Independent Computing Symp.*, USENIX, pages 91–97, April 1995.
- [9] J.J. Kistler and M. Satyanarayanan.  
Disconnected Operation in the Coda File System.  
*ACM Trans. Computer Systems*, 10(1):3–25, February 1992.
- [10] P. Kumar and M. Satyanarayanan.  
Log-Based Directory Resolution in the Coda File System.

- In *Proc. Second Intl. Conf. on Parallel and Distributed Information Systems*, IEEE, pages 202–213, January 1993.
- [11] P. Kumar and M. Satyanarayanan.  
Flexible and Safe Resolution of File Conflicts.  
In *Proc. USENIX 1995 Tech. Conf.*,  
USENIX, pages 95–106, January 1995.
  - [12] R.G. Minnich.  
The AutoCacher: A File Cache Which Oper-  
ates at the NFS Level.  
In *Proc. Winter 1993 USENIX Conf.*,  
USENIX, pages 77–83, January 1993.
  - [13] L.B. Mummert and M. Satyanarayanan.  
Large Grain Cache Coherence for Intermit-  
tent Connectivity.  
In *Proc. Summer 1994 USENIX Conf.*,  
USENIX, pages 279–289, June 1994.
  - [14] L.B. Mummert, M.R. Ebling, and M. Satya-  
narayanan.  
Exploiting Weak Connectivity for Mobile File  
Access.  
In *Proc. Fifteenth ACM Symp. Operating Sys-  
tems Principles*, ACM, pages 143–155,  
Dec. 1995.
  - [15] B.D. Noble and M. Satyanarayanan.  
An Empirical Study of a Highly Available File  
System.  
In *Proc. 1994 ACM SIGMETRICS Conf. on  
Measurement and Modeling of Computer  
Systems*, ACM, pages 138–149, May 1994.
  - [16] J-S. Pendry and M.K. McKusick.  
*Union Mounts in 4.4BSD-Lite*.  
In *Proc. 1995 USENIX Tech. Conf.*,  
USENIX, pages 25–33, January 1995.
  - [17] J-S. Pendry and N. Williams.  
*Amd — The 4.4 BSD Automounter*.  
Imperial College of Science, Technology, and  
Medicine, London, 5.3 alpha edition,  
March 1991.
  - [18] P. Reiher et al.  
Resolving File Conflicts in the Ficus File Sys-  
tem.  
In *Proc. Summer 1994 USENIX Conf.*,  
USENIX, pages 183–195, June 1994.
  - [19] R. Rivest.  
The MD4 Message-Digest Algorithm.  
RFC 1320, IETF Network Working Group,  
April 1992.
  - [20] R. Rivest.  
The MD5 Message-Digest Algorithm.  
RFC 1321, IETF Network Working Group,  
April 1992.
  - [21] M. Rose.  
Post Office Protocol – Version 3.  
RFC 1225, IETF Network Working Group,  
May 1991.
  - [22] H. Saran and S. Keshav.  
An Empirical Evaluation of Virtual Circuit  
Holding Times in IP-over-ATM Networks.  
In *IEEE Infocom*, IEEE, pages 1132–1140,  
June 1994.
  - [23] M. Satyanarayanan, J.J. Kistler, L.B. Mum-  
mert, M.R. Ebeling, P. Kumar, and Q.  
Lu.  
Experience with Disconnected Operation in a  
Mobile Computing Environment.  
In *Proc. USENIX Mobile and Location-  
Independent Computing Symp.*, USENIX,  
pages 11–28, August 1993.
  - [24] E. Schenk.  
Diald source code.  
[ftp://sunsite.unc.edu/pub/Linux/system/  
Network/serial/diald-0.14.tar.gz](ftp://sunsite.unc.edu/pub/Linux/system/Network/serial/diald-0.14.tar.gz)
  - [25] E. Zadok.  
Amd source code.  
[ftp://ftp.cs.columbia.edu/pub/amd/amd-  
upl102.tar.gz](ftp://ftp.cs.columbia.edu/pub/amd/amd-<br/>upl102.tar.gz)
  - [26] E. Zadok and A. Dupuy.  
HLFSD: Delivering Email to Your \$HOME.  
In *Proc. Seventh Systems Administration  
Conf. (LISA VII)*, USENIX, pages 243–  
254, November 1993.

## A Amd Operation

Amd has many features. The most obvious and commonly used is to mount remote NFS file systems on demand then later, after a period of dis-  
use, to automatically unmount them.

The following is the sequence of relevant events showing how Amd demand-mounts the remote file system `serv:/u/user` the first time that any path on that file system (in this example, the file `/u/user/dir/file`) is accessed.

1. Before Amd is running, directory `/u` does not exist on the local disk.
2. Based on startup instructions, Amd creates `/u` then mounts itself as the NFS server servicing mount point `/u`.

| Lookup | What happens   |
|--------|--|
| /      | return vnode for local /   |
| u      | cross mount point into phony "NFS" f/s; return vnode that contains a made-up file handle meaningful only to Amd and which denotes the "root" of the set of links it is emulating |
| user   | return vnode for symlink whose value is "/amd/serv/u/user"   |
| /      | return vnode for local /   |
| amd    | return vnode for local /amd  |
| serv   | return vnode for local /amd/serv   |
| u      | return vnode for local /amd/serv/u   |
| user   | cross mount point into real NFS f/s; return vnode for root of remote exported f/s  |
| dir    | remote f/s returns handle for directory  |
| file   | remote f/s returns handle for file   |

Figure 3: Example Name Resolution with Amd

Amd is also told at startup that whenever it must mount a remote file system it should do so within the directory /amd. (In this example the convention is that if server something1 exports file system /something2, then it will be mounted as /amd/something1/something2.)

3. When someone accesses /u/user/dir/file, Amd is contacted during the name resolution because it is the process serving mount point /u. Since this is the first access to /u/user, Amd creates the local directories /amd, /amd/serv, /amd/serv/u, /amd/serv/u/user, and /amd/serv/u/user/dir, mounts the remote NFS file system serv:/u/user onto /amd/serv/u/user and thereafter emulates a symlink named /u/user whose value is /amd/serv/u/user. Since /u/user is an emulated path, there is no local disk data structure for it.

4. Access to /u/user/dir/file is then done normally by the kernel's name resolution routine.

First, the kernel calls `getattr(u)` and discovers that /u is a mount point; Amd is the server mounted at that point. Second, the name resolution routine calls `getattr(user)` and is told by Amd that user is a symlink. Next, the name resolution routine calls `readlink(user)` and is told by Amd that the symlink's value is /amd/serv/u/user. At this point the original path /u/user/dir/file has been transformed to /amd/serv/u/user/dir/file. This path contains no symlinks or mount points serviced by Amd, but rather is an "ordinary" path leading to a file on a remote NFS file system; i.e., /amd/serv/u/user is a directory on the local disk, with remote NFS file system serv:/u/user mounted on it. The name resolution algorithm will cross the mount point and return an NFS handle for the remote file.

Figure 3 indicates the result of each lookup operation.

Because Amd is concerned only with name resolution, it implements only the `getattr`, `lookup`, `readlink`, and `readdir` NFS operations.<sup>11</sup>

<sup>11</sup>Amd also implements the `unlink`, `rmdir`, `rename`, and `statfs` operations, but these implementations are very lim-

## B Autocacher Operation

The Autocacher is an old version of Amd plus some changes to implement caching of remote read-only NFS file systems onto the local disk. Autocacher emulates symlinks, like Amd, and it partially emulates files. Specifically, it implements the read operation.

Using the conventions of the example above, when resolving the path `/u/user/dir/file` Amd's only actions are to transform `/u/user` into a symlink with value `/amd/serv/u/user` and to mount `serv:/u/user` on the local directory `/amd/serv/u/user`. Access to `dir/file` is via the remote NFS server. Amd neither emulates the `dir/file` subpath nor makes local disk copies of `dir` and `file`; in contrast, Autocacher does both. Autocacher will create local disk copies of the directory `dir` and the file `dir/file`.

For efficiency, Autocacher will emulate reads against the file. It is common for a file to be looked up but not read; e.g., an `ls` operation. Therefore, Autocacher does not copy a file when it is first looked up, but only when it is first read. Otherwise, executing `ls` on a directory would cause the entire contents of the directory to be cached. When an uncached file is first read, the Autocacher copies the file and services all the read operations from the process that performed the first lookup. The second and later lookups will be directed to the local copy.

More precisely, when Autocacher receives an NFS lookup operation, it reacts in one of three ways:

1. If there is a local copy of the file, Autocacher returns the file handle for the local file.
2. If there is no local copy but there is room on the local disk for the file, Autocacher returns a manufactured file handle that is meaningful only to Autocacher. A later read request for this file handle will cause the Autocacher to copy the file to the local disk and to implement read requests by reading from the local copy.
3. If there is no local copy and there is no room on the local disk for the file, Autocacher also returns a manufactured file handle, but later NFS read requests for this file handle will cause the Autocacher to emulate a symlink pointing to the remote file.

---

ited, serving only to help Amd help mask its presence in the name resolution process.

Just as Amd automatically unmounts recently unused file systems, Autocacher automatically deletes local cached copies of recently unused remote files.

## NOTES







# THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of engineers, scientists, and technicians working on the cutting edge of the computing world. The USENIX technical symposia and system administrator conferences are the essential meeting grounds for the presentation and discussion of the most advanced information on the developments of all aspects of computing systems.

USENIX and its members are dedicated to:

- problem-solving with a practical bias,
- fostering innovation and research that works,
- communicating rapidly the results of both research and innovation,
- providing a neutral forum for the exercise of critical thought and the airing of technical issues.

## SAGE, the System Administrators Guild

The System Administrators Guild, a Special Technical Group within the USENIX Association, is dedicated to the recognition and advancement of system administration as a profession. To join SAGE, you must be a member of USENIX.

## Member Benefits:

- Free subscription to *login:*, the Association's bi-monthly newsletter featuring technical articles, system administration tips and techniques, an international calendar of events, SAGE News, book and software reviews, summaries of sessions at USENIX conferences, Snitch Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts, and much more.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, via the USENIX Online Library on the World Wide Web <<http://www.usenix.org>>.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as object-oriented technologies, security, operating systems, and electronic commerce – as many as seven technical meetings every year.
- Discounts on the purchase of proceedings from USENIX conferences and symposia and other technical publications.
- Discount on purchases of USENIX CD-ROMS
- PGP Key Signing Service (available at conferences)
- Discount on BSDI, Inc. products.
- Discount on the five volume set of 4.4BSD manuals plus CD-ROM published by O'Reilly & Associates, Inc. and USENIX.
- Discount on all publications and software from Prime Time Freeware.
- Savings (10-20%) on selected titles from McGraw-Hill, The MIT Press, Morgan-Kaufman Publishers, Nolo Press, O'Reilly & Associates, Prentice Hall, Uniforum and John Wiley & Sons.
- Special subscription rates to the periodicals *The Linux Journal*, *UniForum Monthly*, *UniNews*, and the annual *UniForum Open Systems Products Directory*.
- The right to vote on matters affecting the Association, its bylaws, election of its directors and officers.

## Supporting Members of the USENIX Association:

ANDATACO  
Andrew Consortium  
Apunix Computer Services  
Crosswind Technologies  
Earthlink Network, Inc.  
Frame Technology Corporation  
ISG Technologies, Inc.  
Matsushita Electrical Industrial Co., Ltd.

Motorola Research & Development  
Open Market, Inc.  
Shiva Corporation  
Sybase, Inc.  
Tandem Computers, Inc.  
UUNET Technologies, Inc.

## SAGE Supporting Members:

Bluestone, Inc.  
Enterprise Systems Management Corporation  
Great Circle Associates

Pencom Systems Administration/PSA  
Southwestern Bell  
Taos Mountain

For further information about membership, conferences or publications, contact: USENIX Association,  
2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA Phone: +1-510-528-8649 Fax: +1-510-548-5738  
Email: [office@usenix.org](mailto:office@usenix.org) URL: <http://www.usenix.org>

ISBN 1-880446-84-7